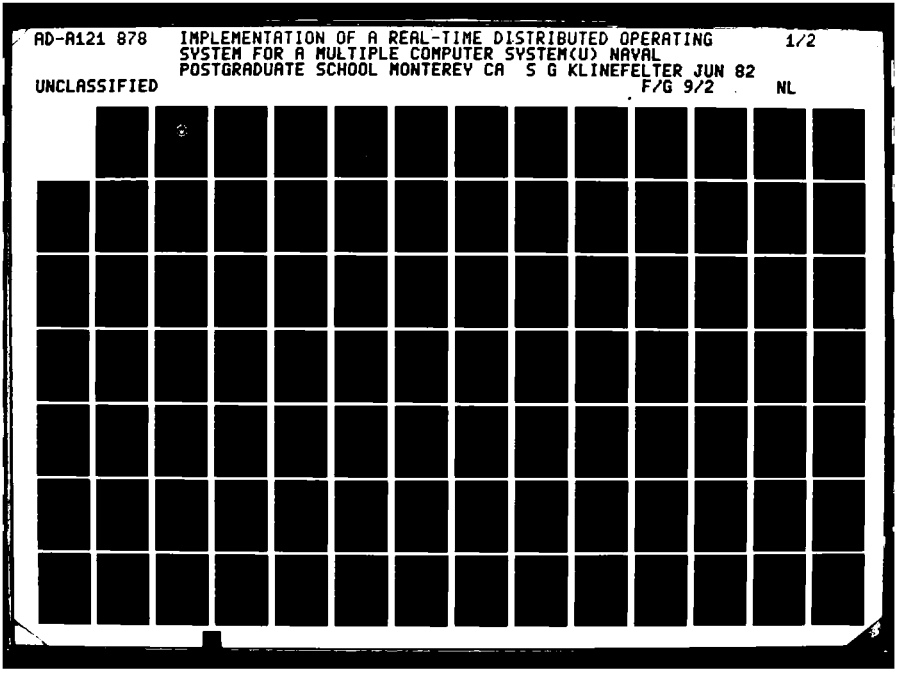
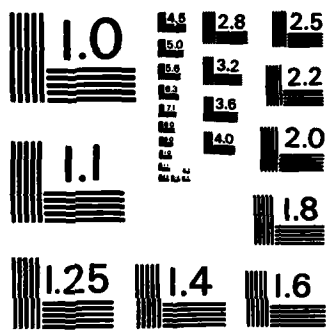

AD-A121 878
IMPLEMENTATION OF A REAL-TIME DISTRIBUTED OPERATING SYSTEM FOR A MULTIPLE COMPUTER SYSTEM(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA S G KLINEFELTER JUN 82
1/2
UNCLASSIFIED
F/G 9/2
NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

IMPLEMENTATION OF A REAL-TIME,
DISTRIBUTED OPERATING SYSTEM
FOR A MULTIPLE COMPUTER SYSTEM

by

Stephen G. Klinefelter

June, 1982

Thesis Advisor: Dr. Uno R. Kodres

Approved for public release, distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A121878	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Implementation of a Real-time, Distributed Operating System for a Multiple Computer System		5. TYPE OF REPORT & PERIOD COVERED M. S. Thesis June, 1982
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Stephen Graham de Grimaldi Klinefelter		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June, 1982
		13. NUMBER OF PAGES 171
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Operating system, microcomputers, real-time processing, multiprocessing, distributed computer networks

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis presents extensions to an implementation of a kernel in a real-time distributed operating system for a microcomputer based multiprocessor system.

The operating system, MCORTEX, is a 2 level, hierarchically structured loop free, system that permits logical distribution of the kernel in the address space of each process. The design is based on segmented address spaces and per process stacks. Process synchronization is achieved through sequencers and eventcounts. MCORTEX is resident in the local memory of each microcomputer but

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

system data is maintained in shared global memory.

MCORTEX has been extended to include a system monitor process which allows stopping the system to examine any memory, shared or local, from any location. The system can then be restarted without reinitializing each micorcomputer.

This system particularly supports applications where jobs are partitioned into a set of multiple interacting asynchronous processes. The system is currently implemented on INTEL 86/12A single-board computers.

A

Approved for public release; Distribution unlimited

Implementation of a Real-time, Distributed Operating System for a Multiple Computer System

by

Stephen Klinefelter
Captain, United States Army
B.S., Virginia Military Institute, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June, 1982

Author: Stephen G. Klinefelter

APPROVED BY: Uno R. Kodres
Thesis Advisor

Ronald Mack
Second Reader

Chairman, Department of Computer Science

W. M. Woods
Dean of Information and Policy Sciences

ABSTRACT

This thesis presents extensions to an implementation of a kernel in a real-time distributed operating system for a microcomputer based multi-processor system.

The operating system, MCORTEX, is a 2 level, hierarchically structured, loop free, system that permits logical distribution of the kernel in the address space of each process. The design is based on segmented address spaces and per process stacks. Process synchronization is achieved through sequencers and eventcounts. MCORTEX is resident in the local memory of each microcomputer but system data is maintained in shared global memory.

MCORTEX has been extended to include a system monitor process which allows stopping the system to examine any memory, shared or local, from any location. The system can then be restarted without reinitializing each microcomputer.

This system particularly supports applications where jobs are partitioned into a set of multiple interacting asynchronous processes. The system is currently implemented on INTEL 86/12A single-board computers.

DISCLAIMER

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempt to cite each individual occurence of a trademark, all registered trademarks appearing in this thesis will be listed below, following the firm holding the trademark:

INTEL Corporation, Santa Clara, California

INTEL	MULTIBUS
iSBC 86/12A	INTELLEC MDS
ISIS-II	PL/M-86
8086	MCS-86

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. GENERAL DISCUSSION

This thesis presents extensions to an implementation of a kernel in a real-time distributed operating system for a microcomputer based multi-processor system, called MCORTEX.

As the performance and capabilities of micro-computers continues to improve, it is becoming apparent that many large real-time applications that are managed by large fast mini-computers could be managed by an integrated multi-processor system of less expensive, commercially-available micro-computers. What a single general purpose micro-computer might lack in speed could be compensated for by a system of computers if it is managed by an operating system that allows process synchronization and parallel or concurrent processing.

There are multi-microcomputer systems already in use today in real-time applications. However, in order to accommodate high data input rates and the addition of more processors to the system, custom interconnections often have to be devised so that system performance is not degraded. These custom designs may increase the cost of the system and reduce flexibility in meeting the needs of a variety of applications.

The purpose of this thesis is to continue development of an operating system design that is simple, small, and flexible. A principal goal is to demonstrate the operating system on commercially available, relatively inexpensive general purpose micro-computers requiring the minimum of custom-developed hardware for processor inter-communication and control. It is anticipated that the design will be general in nature in order to be applied towards various real-time applications and implemented on different micro-computer systems with the minimum of modification. The specific goals of this thesis are discussed in the next section concerning the background of the project.

B. BACKGROUND

The AEGIS weapons system simulation project currently being conducted at the Naval Postgraduate School is attempting to determine the feasibility of raplacing much of the larger and relatively expensive mainframe computer, the AN/UYK-7, with a system of 16 or 32 bit micro computers. Several significant real-time functions of the AEGIS Weapons System are to be duplicated with associated data, inputs, timing, and supporting functions so that a test example can be examined whose performance emulates that of the actual system.

In particular, emulation of the SPY-1A radar, which is a sub-system of AEGIS, is being examined. Signal processing

for long distance low altitude missile detection and target acquisition for tactical missiles requires processing large amounts of collected data in real-time. It is proposed that a system of micro-computers as described above could provide the processing power required to perform concurrent asynchronous computations.

Design of the operating system that would manage such micro-computer system was started by Wasson who defined the detailed design of an operating system tailored to real-time image processing[Ref. 10]. He based his design on a more general model developed by O'Connel and Richardson of the Naval Postgraduate School in 1979. The design was to be applied to the general purpose INTEL iSBC 86/12A micro-computer. This single board micro-computer is based on the 16 bit INTEL 8086 microprocessor. Wasson's design used the MULTICS concepts of segmentation and per process stacks and Reed and Kanodia's enventcount synchronization methods.[Ref. 9: pp.12-13] Rapantzikos began the implementation of Wasson's design[Ref. 11].

The operating system, MCORTEX, at this point used the concept of a "two level traffic controller" to accomplish processor multiplexing among a greater number of eligible processes. This dual-level "processor multiplexing design" allowed the system to treat the two primary scheduling decisions, viz., the scheduling of processes and the

management of processors, at two separate levels of abstraction.

Cox continued the implementation effort of Rapantzikos by greatly simplifying the design of MCORTEX. He placed a higher priority on shortening the execution time in MCORTEX over the possible added security of a two level traffic controller and therefore, reduced the traffic controller to one level of abstraction which simplified the implementation. His other contribution was to add a gatekeeper module to the top of the operating system so that operating system calls were made through a single "gate" and so that the user would not have to concern himself with service codes. The result was a very compact, trimmed down, basic operating system which supports multiprocessors performing multiprocessing. [Ref. 12: pp.13-14] Cox demonstrated MCORTEX with a system of three user processes executing concurrently on two INTEL iSBC86/12A microcomputers connected by the INTEL MULTIBUS.

The specific goals of this thesis are to:

1. Test the generality of MCORTEX by expanding the system to several additional micro-computers.

2. Expand the system to the extent necessary to provide a means of dynamically interacting with the operating system while it is executing so that efficient testing of the multiprocessor system can be accomplished. Previously, MCORTEX had to be allowed to "run its course"

before examining various structures in system memory. It then had to be completely reloaded and reinitialized to resume execution.

C. STRUCTURE OF THE THESIS

Chapter I presents a general discussion of the larger, ongoing effort of which this thesis is a part. It also gives a general discussion of the background work that has been accomplished regarding the specific topic of this thesis, the MCORTEX operating system.

Chapter II addresses the overall design philosohpy of MCORTEX and its functional requirements. Multiple process communication and synchronization tasks and techniques are included. There is also a discussion of process multiplexing and other utilities support for the user. However, the discussion is limited, inasmuch as three prior theses have devoted their attention to the general design.

Chapter III describes the hardware architecture of the system on which MCORTEX is demonstrated and why this particular micro-computer was chosen.

As the iSBC 86/12A is a developmental system, the first part of Chapter IV outlines the INTEL corporation support systems that make the use of the iSBC 86/12A possible. Chapter IV then addresses the detailed design of MCORTEX. Testing of the previous version, explanations for the path

chosen to implement enhancements, problems encountered, and their implications are discussed.

Chapter V summarizes the testing of the operating system and describes the new capabilities available to the user. Suggestions are also given for future research and testing.

II. BASIC DESIGN CONCEPTS

A. PROCESS STRUCTURE

Dividing a job into asynchronous parts and concurrently executing these parts as separate entities offers significant advantages if the job is continuously receiving input and/or processing data. In a single processor the benefits are mainly confined to design simplicity. However, in a multi-processor system these asynchronous parts, or processes, are essential if the system is to take advantage of parallel and pipeline processing.

There are two elements that, together, are sufficient to characterize a process; (1) the process address space and (2) its execution point.

The address space is the set of memory locations that could be accessed during process execution. There exists a possible path to all the memory locations in that space during the life of the process. It is this important characteristic that allows the "distributed" operating system to be viewed as part of the address space of the process. The entire address space can be viewed as having two domains of execution, the user domain and the kernel domain. Entry to the kernel domain of execution is additionally restricted to a single entry point.

This enhances design and ensures a certain measure of security. See Fig. 1.

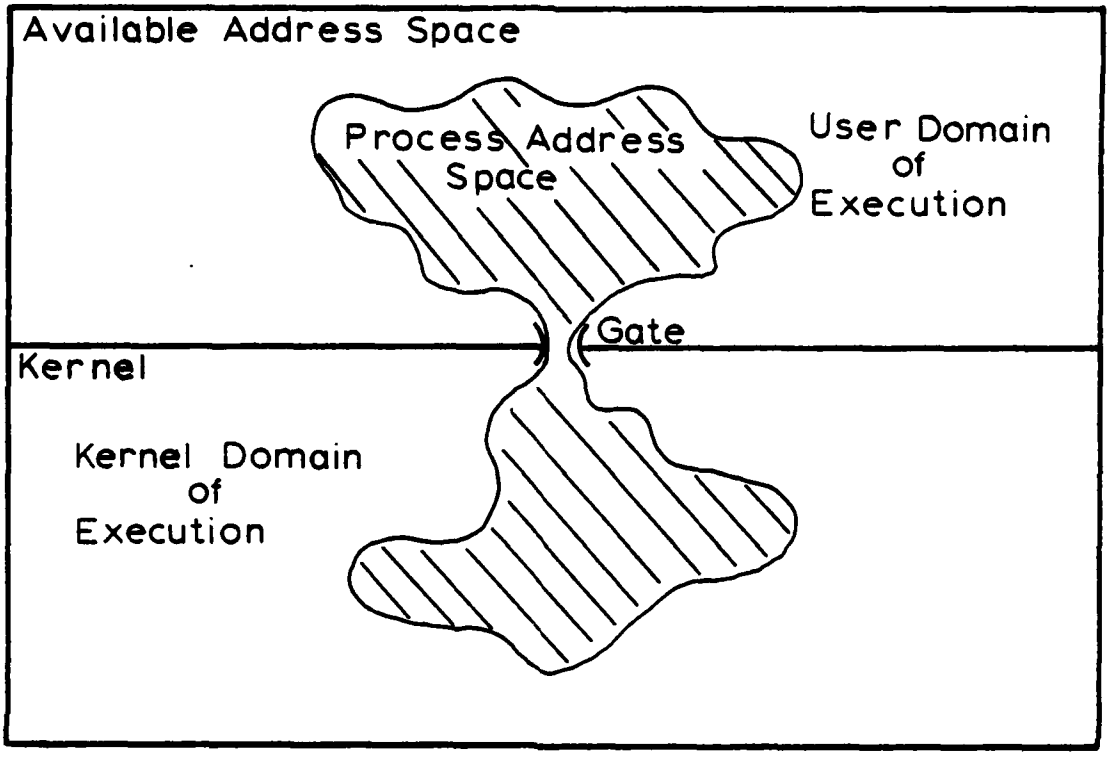


Fig.1 Address Space

The execution point is characterized by the state of the machine at any particular time and consists of the contents of certain machine registers.

By designing a system as a collection of cooperating processes, system complexity can be greatly reduced. The asynchronous nature of the system can be structured logically by representing each independent sequential task as a process and by providing interprocess synchronization and communication mechanisms to prevent race and deadlock situations during process interactions.

If the processes are confined to well-defined address spaces that do not overlap then they would never interfere with each other. Some controlled form of overlapping of address spaces or sharing must exist if there is to be, as a minimum, coordination between the processes.

B. VIRTUAL PROCESSORS AND SCHEDULING

A virtual processor is an abstraction. It is a data structure that contains all the required data that describes the execution point of a process at any given instant. There is a virtual processor for every process but only one real processor. Each real processor has up to a fixed number of virtual processors and each virtual processor has an "affinity" for a particular real processor. This will be explained later in terms of the Virtual Processor Map.

The virtual processor also has use of the process stack which contains the procedure activation records and other necessary data for the execution of the process. In actuality the data structure that represents the virtual processor contains a pointer to this stack. Other data that represents the virtual processor are "priority" of the process it is executing, current state of the virtual processor, and any "event" the virtual processor might be waiting for.

As all virtual processors have the same components, the data structures that represent the virtual processors form

an array or structure called the virtual processor map (VPM). The VPM is used by the scheduler to select the proper eligible process to be executed. After selection, virtual processor's stack is used to restore the real processor to the state when it was last executing that process. This selected virtual processor is now "running" as opposed to "waiting" or being "ready" to run. The real processor, which is a physical object, is always running. In this case the real processor is the INTEL 8086.

It is the abstraction of the real processor that allows multiplexing of several processes on a single processor resource. The high level system calls of the operating system operate on these virtual processors thus making the design independent of the configuration of the hardware. Adding real processors to the system, up to the bandwidth of the system bus, would not affect the user except that he is likely to obtain increased performance.

User services available via procedure calls through the gate can be regarded as an extended instruction set which is available to the virtual processor but not the real processors. Rapantzikos uses, as an example, the "Await" operation. It does not use real processor resources but it does inhibit the use of the real processor by the virtual processor. [Ref. 11: pp.43-44]

C. MULTIPLEXING

Multiplexing could also be called multiprogramming and it is facilitated by the abstraction of the real processor described in the previous section. The execution by the virtual processor of a sequence of operations is actually performed by the real processor. However, the sequential operations may be separated by gaps of time when other virtual processors are executing a sequence of operations. This allows efficient use of scarce real processor resources. Figure 2 shows how several virtual processors could be mapped into the operation sequence of one real processor. When each virtual processor is executing on the real processor it said to be "bound" to the real processor. The length of each horizontal line represents the amount of time that virtual processor is bound to the real processor. Only one virtual processor can be running at any point in time. However, any number of virtual processors can be ready-to-be-scheduled or waiting for some other event. Figure 3 shows the possible state transitions of a virtual processor.

For a portion of the time that a virtual processor is executing, it could be in the kernel domain of execution. This implementation uses a distributed kernel which is interruptable and loop free. Thus, the virtual processor can be interrupted at almost any point in time and continue on the next time it is scheduled. It should be noted that

loop free indicates a condition in which a module which uses part of a second module cannot in turn be used or called by that second module.

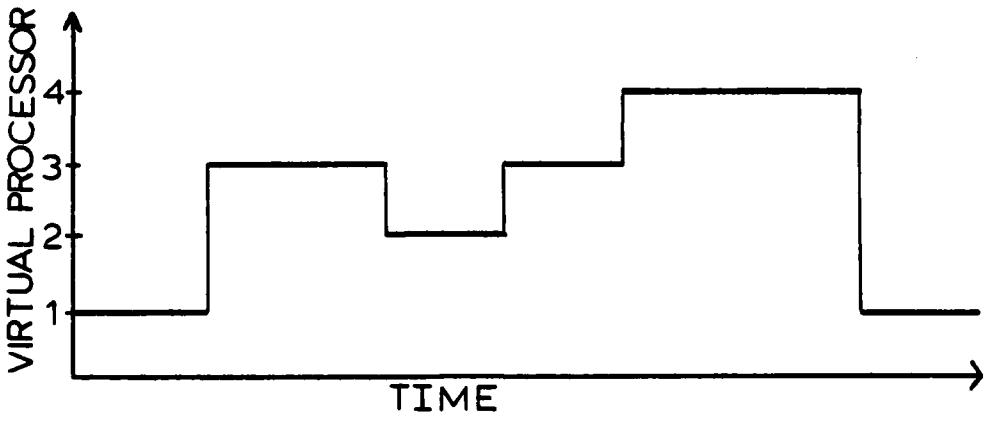


Fig. 2 Multiplexing

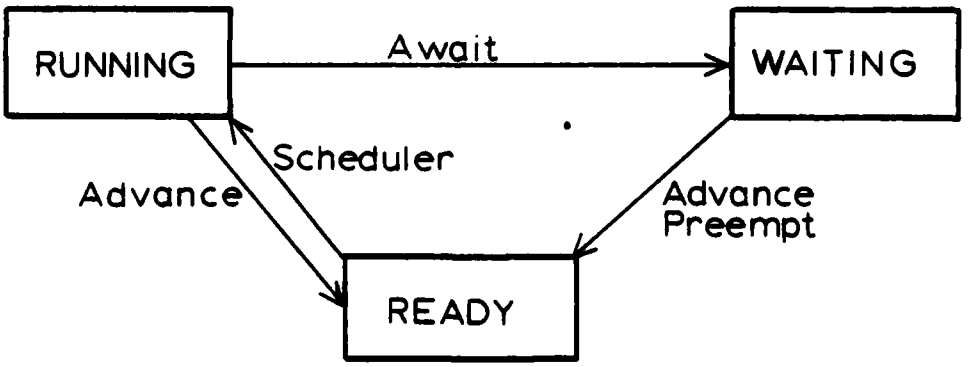


Fig. 3 Virtual Processor State Transitions

Therefore, multiplexing is the means by which a system which has more processes than it has processors, can efficiently schedule those processes and make effective use of the real processor's time. It also makes the design and

implementation of the operating system easier to understand and document.

D. MULTIPROCESSING

In a multi-processor environment, concurrent processing is a natural byproduct. The process structure previously revealed is used to divide a job into separate sequential tasks that can now be scheduled to run concurrently. Although the response time for a job might not decrease, the throughput would increase. Therefore, significant gains in efficiency can be made.

When multiplexing and multiprocessing are combined, a system is obtained with a significant throughput advantage that can react efficiently to asynchronous events. This implementation uses a single, system-wide, preemptive interrupt mechanism to signal the need for rescheduling in the multi-processing environment. Furthermore, the above requirements reinforce the choice of the iSBC86/12A to implement MCORTEX. Commercially-available, general-purpose hardware is available to interconnect the iSBC86/12A micro-computers.

It should be noted that in the interest of keeping MCORTEX small and fast, virtual processors being scheduled on one real processor will never be scheduled on another. This is a result of Cox's work in symplifying the design of MCORTEX. Virtual processors have an "affinity" for the real

processor on which they were originally loaded. It must be remembered that MCORTEX is for a real-time multi-processor environment and scheduling decisions must take the minimum time possible. The memory management functions that would be required otherwise would degrade the performance of the real-time environment.

E. COMMUNICATION AND SYNCHRONIZATION

For concurrent processing, a job that is composed of sequential and non-sequential tasks is explicitly divided into an appropriate structure of processes that can run concurrently. Inter-process communication and synchronization are necessary for concurrent processing.

It is the responsibility of the operating system to provide mechanisms for effective communication between cooperating processes. There are two kinds of communications that processes must be able to achieve. There must exist a way for processes to exchange data. This is called inter-process communication. There must also exist a method for processes to order their execution in response to certain events, particularly events affecting the shared memory status and validity. Therefore, to utilize the parallelism and pipelining afforded by multiple processors, a mechanism is required for synchronization between processes. Rapantzikos [Ref. 11] made the decision to use the eventcounts and sequencers of Reed and Kanodia [Ref. 9].

This provides automatic "broadcasting" and supports "parallel signaling". Another reason eventcounts and sequencers were chosen is because in addition to providing synchronization, a history of the jobs execution is maintained.

The synchronization between processes is supported by the "Await", "Advance", and "Preempt" functions. Except for "Preempt", these functions operate on the eventcounts. The "Ticket" function operates on sequencers. Cox [Ref. 12:p.26] mentions that the "Ticket" function applied to sequencers supports the exclusive access of a process to a shared resource. It is also used to provide odered access to the resouce on a first come, first served basis.

The number of times an "event" has occurred is represented by an eventcount. Await blocks the currently running process until a threshold value has been reached for a particular eventcount. "Advance" increments the current value for a particular eventcount. Thus, "Await" and "Advance" can be used to provide cooperation between processes.

"Preempt" forces the scheduling of a high priority process that will then block itself. This is sometimes required for very critical asynchronous processes. However, if there still exists higher priority processes that are ready, they will be scheduled first.

"Preempt" is used sparingly and only for the very highest priority processes normally.

III. MULTIPROCESSOR ARCHITECTURE

The micro-computer selected to initially implement MCORTEX is the INTEL iSBC86/12A single board computer (SBC). It is based on the INTEL 8086 16 bit micro-processor. Detailed descriptions of all the components of the SBC and the multiprocessor interface, the MULTIBUS, can be found in [Ref. 1] and [Ref. 2].

A. HARDWARE REQUIREMENTS

1. Shared Global Memory

Shared global memory is required in this implementation as the means for virtual processors to coordinate their communication with each other. The operating system is resident in the local memory of each real processor where it is a distributed part of the address space of each process associated with that processor. However, data concerning all the virtual processors in the multi-processor system exists as one copy in shared global memory. One of the reasons for keeping all virtual processor data in one structure in global memory is the way the total structure is operated on by various MCORTEX synchronization system calls. Those functions as well as the global data base structure will be detailed in Chapter IV, Section C.

2. Synchronization Support

To prevent concurrency problems in the shared global memory, a software lock is used for access to the shared data base. However, the lock itself has to be protected from deadlock and race conditions. Therefore, an indivisible "test and set" mechanism is required. PL/M-86 and ASM86 both support this protection device. In PL/M-86, the mechanism is the built-in procedure "Lockset" [Ref. 8:pp.12-14 to 12-15]. In ASM86 support comes in the form of an instruction prefix "lock" [Ref. 5:p.5-94]. The "lock" prefix, which is also used in the function "Lockset", causes the system bus, the MULTIBUS, to be locked during the duration of the following instruction. This prevents the interleaving of instructions and/or access by another processor. Therefore, only one processor can access and change the global lock variable at any one time if the above support feature is utilized. By necessary convention, it is the responsibility of the processor who last had access to the lock variable to unlock it. Processors are then assured of mutual exclusion while updating the shared data base. It is worth reiterating that the system bus is locked only during the testing and setting of the software lock and not during the software restricted access to the shared data base. Normal use of the system bus continues while one real processor is accessing that part of shared memory protected by the software lock.

3. Inter-processor Communications

Some method of communication between real processors is required if the processes they are executing are going to be synchronized. In the asynchronous real-time environment, the communication will invariably come in the form of an interrupt mechanism. The iSBC86/12A supports many different interrupt schemes. This implementation requires a single system interrupt that any real processor can use to "broadcast" an event. Every other real processor has to be able to recognize it and react to it. This preemptive interrupt signals to the other processors that a possible rescheduling of a virtual processor is necessary. The specific interrupt structure in hardware is detailed in this chapter in Section D. The software mechanism that decides which processor should react to the broadcast interrupt signal is detailed in Chapter IV, Section E.

B. HARDWARE CONFIGURATION

1. System Configuration

Figure 4 shows a general drawing of the multi-processor system configuration. The CPU's are iSBC86/12A single board micro-computers that come with 64K bytes of local memory. In Cox's work an inactive iSBC86/12A's memory served as the global memory module [Ref. 12:p.49]. The module was replaced in this implementation with a 32K byte RAM board compatible with MULTIBUS. Communication with each

CPU for the purpose of loading developed software is via the serial port on each SBC.

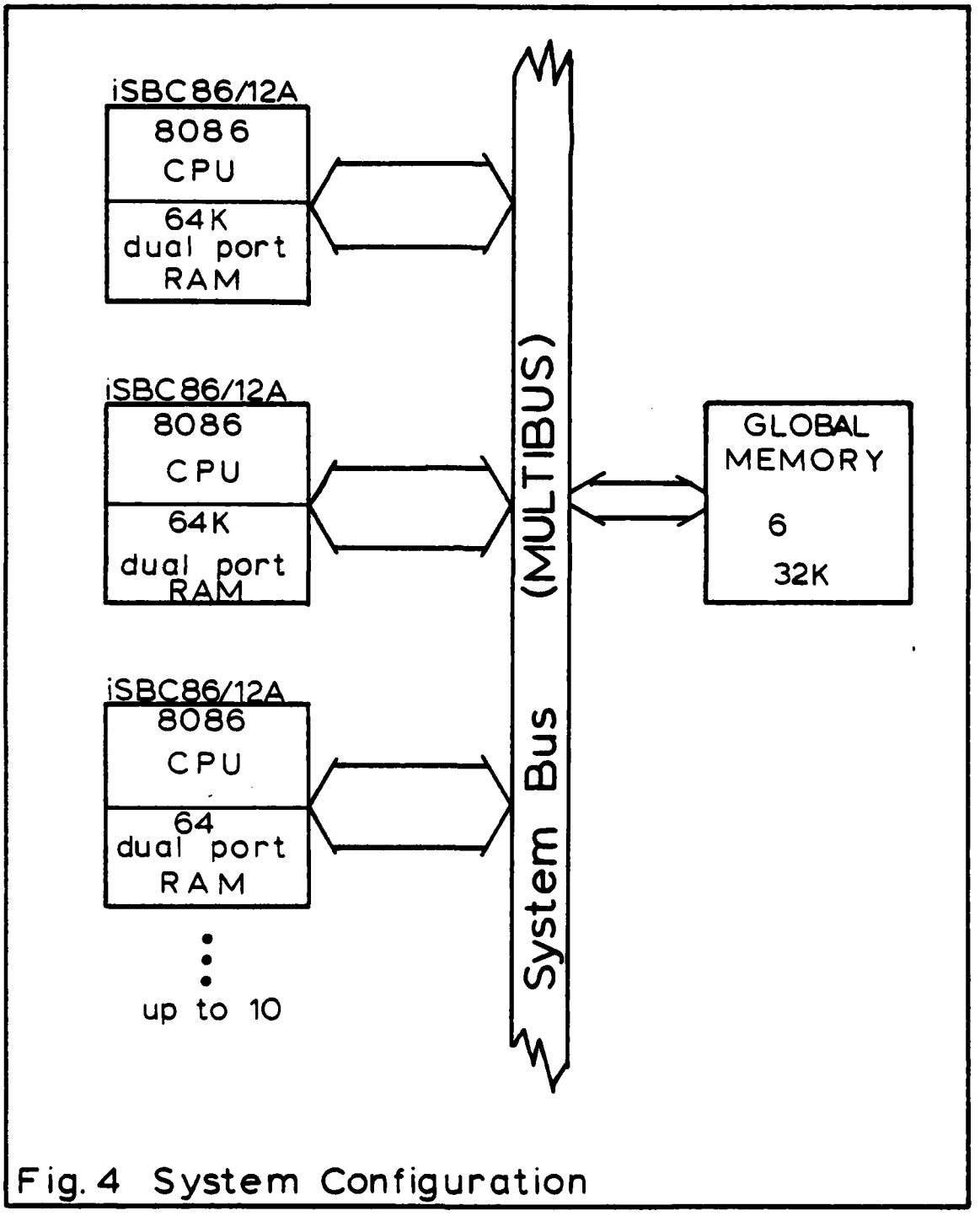


Fig. 4 System Configuration

2. The 8086 Microprocessor

The 8086 is well-documented in [Ref. 1] and [Ref. 2]. This section is intended to give general knowledge about the 8086.

The 8086 is a high performance, general purpose microprocessor. The CPU contains an Execution Unit (EU) and a Bus Interface Unit (BIU). The two units operate independently of each other. The EU executes instructions in much the same way as a conventional CPU. But the BIU is dedicated to fetching instructions, reading operands, and writing operands which the EU is executing or operating on respectively. The BIU also "pipelines" or stacks instructions in an internal RAM array to a level of six. Thus, a majority of the fetch time in executing instructions disappears. The number of instructions executed per unit time significantly increases and idle time on the bus is minimized.

The 8086 has a 16 bit data path and address space of one megabyte. The CPU gives direct hardware support to programs written in the high level language, PL/M-86. Those very low level routines that must be written in assembly language are developed in ASM86. The object code modules of each can be linked together without difficulty. The basic instruction set provides for direct operations on memory, to include stack operands. It provides software generated interrupts, byte translations from one code to another,

move, scan, and compare operations for strings up to 64K bytes in length, multiplications and division of binary numbers.

The 8086 has eight 16 bit general registers, four of which are byte addressable. The remaining four are index or pointer registers, which could also be utilized as accumulators like the first four. Figure 5 shows the organization of the 8086 registers and their principal use.

The 8086 has nine status bits that can be set in the flag register which is a 16 bit register. The functions of the status and control bits are listed in Figure 5. One bit, the IF flag or interrupt flag, has special significance. If the flag is "1" it enables maskable interrupts. If it is "0" it disables maskable interrupts. The flag must be initialized for appropriate use in the operating system. The IF flag will be described in greater detail in a later section.

3. Segmentation and Segmentation Registers

The 8086 does not support the notion of explicit segmentation. For example, it contains no special hardware to perform memory management functions, segment access checks, or bounds checking. However, addressing is "segment-like" as it is two- dimensional. All addresses are composed of two parts: the base and the offset. 8086 programs view the one megabyte address space of the 8086 as a group of segments that are defined by the application.

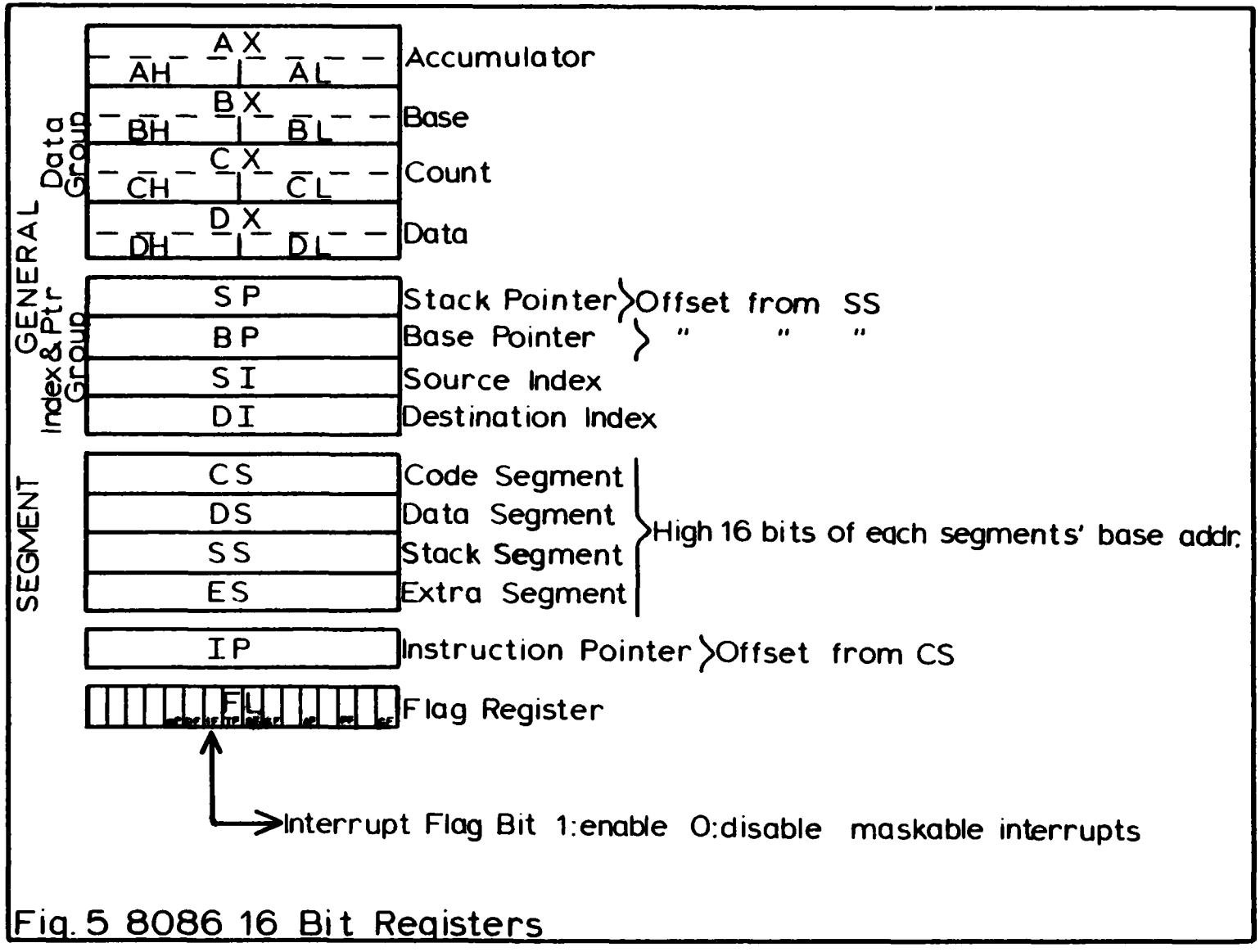


Fig. 5 8086 16 Bit Registers

The base is an absolute address that points to the low memory end of a segment aligned on 16 byte boundaries. The offset is a 16 bit number that indicates position from the base address. Therefore, any segment can be up to 64K bytes long and consists of contiguous memory. The segment base address can point to any location in the one megabyte address space. As the base segment address can only be 16 bits long, the low four bits are assumed to be zero. Segments may overlap, partially overlap, be adjacent to each other, or be disjoint.

The segment registers contain the various segment base addresses. At any point in time the CPU can address four segments. The function of the segment registers may sometimes alternalte but in this implementation they are used in their default roles. Figure 5 applies to the next several paragraphs.

The code segment register (CS) contains the high 16 bits of the absolute address of the currently executing code segment. The register is used in conjunction with the instruction pointer register (IP) which is an offset value from the contents of the CS register. Together, these registers are used in a similar manner to the program counter of smaller micro-processors. All instructions are fetched from this segment and the IP always points to the next instruction to be fetched.

The data segment register (DS) contains the starting address of the current data segment and generally contains program constants and strings. It is routinely used in conjunction with the source index register (SI) and the destination index register (DI). The DS register may be changed during execution of the program and, normally, each code segment will have an accompanying data segment.

The stack segment register (SS) is used to implement the per process stacks required by MCORTEX. It is used in conjunction with the stack pointer register (SP) and the base pointer register (BP). Rapantzikos had two stacks in effect at any one time in his implementation, a kernel stack and a user stack [Ref. 11:p.74]. When Cox simplified MCORTEX, only one stack remained. As MCORTEX can be viewed as part of the process address space, there is only a need for a per process stack. The contents of the SP register indicate an offset in higher memory from the SS register contents. SP points at the current top of the stack which grows towards lower memory and the stack base. Therefore, the initial value of SP indicates the bottom of the stack. BP is kind of a utility marker. The first time it is used, it points to the same location as SP at the beginning of the stack. If for some reason, a group of arguments are pushed on the stack (as in preparation for a procedure call), the BP contents are pushed on the stack and then BP is updated to equal the current SP (top of the stack). In this way, an

activation record is delineated. When it comes time to discard the arguments, the CPU knows how far back down the stack to go. (In actuality, the SP is updated with the BP value on top of the stack. Nothing is ever erased from the stack. SP and BP are simply constantly manipulated. Growth of the stack will overwrite some locations.) Stacks can be up to 64K bytes long and, like all other segments, can be placed anywhere in the one megabyte address space.

The extra segment register (ES) is typically used for external or shared data, and data storage.

4. The iSBC86/12A(Single Board Computer)

The iSBC86/12A is a complete computer capable of stand-alone operation. It is used as the basic processing node of this multiprocessor system. The SBC includes the 16 bit 8086 CPU, 64K bytes of RAM, a serial communications hK9terface, three programmable parallel I/O ports, programmable timers, a programmable interrupt controller, MULTIBUS interface logic, and bus expansion driver for interface with other MULTIBUS compatible expansion boards. There are provisions for the installation of up to 16K bytes of ROM. As MCORTEX progresses it is anticipated that the operating system will be programmed into ROM.

There are three sets of connections that must be made to each SBC. By convention one single board computer, SBC #1, has been selected to provide the MULTIBUS clock signal. Switches and jumpers for extended addressing must

be set and there are three jumpers used to implement the preemptive interrupt system. The system bus clock is provided by jumping the adjacent pins E105 and E106 on SBC#1 ONLY. The latter two sets of connections are explained in the next two sections.

C. PHYSICAL ADDRESS GENERATION

1. General

There are two methods to specify a memory address. One is the absolute or physical address and the other is the logical address. There is only one physical address that will specify a given memory location but there may be several logical addresses equivalent to one absolute address. Absolute addresses can range from 0 to FFFFFH. The following are two examples of how the processor reconstructs absolute addresses from two-dimensional logical addresses:

a. 1234:0000 b. 1200:0340

It should be noted that the base and offset are held in separate 16 bit registers. The base in both examples is shifted left four bits and then the offset is added. A 20 bit address results which ranges the entire one megabyte address space.

a.		b.	
	1234		1200
	12340<		12000<
	+0000		+0340
	12340		12340

The shifting of the base is what forces the alignment of

segments on 16 byte boundaries. It is apparent that the two different logical addresses are equal to the same absolute address. The above is an extended address because it falls outside the range 0 to FFFFH or 64K byte range of a single processor. Which processor the reference is made to is discussed in the next section.

2. Local and Extended Addresses

Addresses outside the range o to FFFFH are extended addresses from the point of view of a single board computer. As the other SBC's also have 64K bytes of local RAM, there must be some way of designating where in the one-megabyte address space they fall. Locally, the SBC does not need or recognize the 20 bit address. If one is specified, the SBC has to know, via the MULTIBUS, where to go. The iBSC86/12A contains jumpers and switches that allow it to translate 20 bit addresses into local memory. If each SBC is configured properly, each 64K byte local RAM could be given a unique identity in the one- megabyte address space as viewed from the MULTIBUS. Those connections are listed in the next section. Understanding how local and extended addressing works, and how absolute addresses are formed from two-dimensional addresses is important because the implementation of MCORTEX on the ISBC86/12A depends on it. The next several paragraphs assume that the SBC's have been configured as listed above and that the full 64K bytes of RAM is visible or accessible from the MULTIBUS.

From the point of view of the monitor currently resident in ROM on the SBC, the designer can specify absolute or two-dimension addresses. For example, the display command, "D", can be executed by typing:

a. .D2345 or b. .D0200:0345

The many ways Example (b) could be listed have already been explained in Section A. Example (a) will return the same result. The difference is noted in how the CS and IP registers will be loaded to fetch the contents at that location which, incidentally is in local memory. If Example (a) had been a number greater than FFFFH, the address would have been "wrapped around" to lower memory. For example:

.D12345

would have returned the contents at 2345H, as that was loaded into the SI. DS was assumed to be zero. However, an example such as:

.D1200:0345

would have returned the contents at 12345H, which is external to the local memory. In this implementation, the address would have been on SBC #1 at its local absolute address of 2345H. SBC #1 as viewed from the MULTIBUS, has all of the memory from 10000H to 1FFFFH. A full listing of these translations will be given in the next section.

From the point of view of the HOL, PL/M-86, knowledge of local extended addressing is also important. The compiler represents pointer values differently depending

on options set at compile time. All code compiled for MCORTEX and the user processes must be done with the attribute "LARGE". For example:

```
:F1:PLM86 PROCESS5.SRC LARGE
```

will result in the PLM source file "PROCES5.SRC" to be compiled. The attribute "LARGE" indicates to the compiler that references outside the 64K byte range of local memory are to be made, and therefore, address references have to be represented in a certain way. Specifically, pointer variables can now have the value 0 to FFFFFH and even if the source code contains an absolute reference, it will be represented in memory as two words. The higher two bytes will contain a base value from 0 to FFFFH and the lower two bytes will contain an offset value from 0 to FFFFH. For both the offset and base the least significant 8 bits are in the first byte and the most significant 8 bits are in the second byte. This knowledge was used to implement the diagnostic monitor process so that it could access any memory from any SBC in the multi-processor system. [Ref. 6:pp.B-1 to B-8] [Ref. 7:p.5-4 and 8-1] [Ref. 8:p.4-14 and 8-3]

The locator, "LOC86", also makes use of local and extended addresses. The locator takes an object module that has been linked with necessary modules and resolves all relative referances into actual addresses. To maintain the integrity of modules during testing and because it is

necessary to overlay some modules on top of each other, certain segments are "located" at specific addresses. The following example comes from the locate command file actually used to locate the operating system whose file name is "KORE.LNK":

```
LOC86 KORE.LNK ADDRESSES(SEGMENTS(&
STACK(03000H),&
INITMOD CODE(02800H),&
GLOBALMODULE DATA(0E0000H)))&
SEGSIZE(STACK(75H))&
RS(0H TO 0FFFH)
```

Absolute addresses have been specified for two modules and a stack segment. All will be located at local addresses except for the global data base. An external 32K RAM board has been attached to the MULTIBUS to serve as global memory. It will recognize references from the MULTIBUS from E0000H to E7FFFH. The output of this command is the executable code module, KORE, that could then be loaded on all the SBC's. Each SBC would have a copy of MCORTEX in its local memory and each would "know" where to go to find the global data base.

3. Hardware Connections for Local/Extended Addressing

The following are the hardware connections needed to set up RAM addresses properly for MULTIBUS interface access. They consist of one jumper connection and eight switch settings on a dual-inline package (DIP) called S1. Each board's full 64K byte RAM is made accessible to the MULTIBUS by setting switches #5 and #6 open on every SBC. Switch #8

on S1 is always open. Switches 1-4 have to do with displacement of the base address as seen from the MULTIBUS. It is necessary to only use switch #1 to indicate a 64K displacement from the top of a selected 128K byte segment in the one-megabyte address space. Switches 3-4 are always open. The jumpers select the 128K byte segment of the one-megabyte address space where the 64K byte RAM will be placed. [Ref. 2:pp.2-7 to 2-9]

Table 1. RAM ADDRESSES(MULTIBUS INTERFACE ACCESS)

SBC #	Switch #1	Jumper Connections	Placement
0	closed	127-128	00000H-0FFFFH
1	open	127-128	10000H-1FFFFH
2	closed	125-126	20000H-2FFFFH
3	open	125-126	30000H-3FFFFH
4	closed	123-124	40000H-4FFFFH
5	open	123-124	50000H-5FFFFH
6	closed	121-122	60000H-5FFFFH
7	open	121-122	70000H-7FFFFH
8	closed	119-120	80000H-8FFFFH
9	open	119-120	90000H-9FFFFH

D. INTERRUPT HARDWARE

1. Description

As with most other micro-processors, the 8086 does not possess the capability to directly generate interrupts destined for other devices. This characteristic is needed to implement preemptive scheduling. One of the high bits from one of the parallel ports is used to initiate the interrupt. The MULTIBUS has eight interrupt request lines that can be used, INTR0 through INTR7. INTR4 was arbitrarily selected. The output bit from the parallel port must drive

the interrupt request line and it must follow convention by using a negative-going-signal on the MULTIBUS interrupt request line to request interrupt service. A single input nand gate serves to drive and invert the signal onto the MULTIBUS. Likewise, the same interrupt request line must be connected to the programmable interrupt controller (PIC) which recognizes a positive or positive-going-signal to initiate an interrupt sequence with the CPU. By using the single INTR4 line, all SBC's react to the interrupt signal when one board issues it (as well as the one that issued it.) This does not create a problem and it will be explained in the next chapter. In summary, the high bit from a parrallel port is used to broadcast an interrupt signal onto the MULTIBUS INTR4 line which is in turn connected to the PIC.

2. Hardware Connections

Figure 6 shows three connections that must be made:

E9 to E13
E69 to E77
E137 to E142

More detailed drawings are contained in [Ref. 2:pp.5-21 to 5-24].

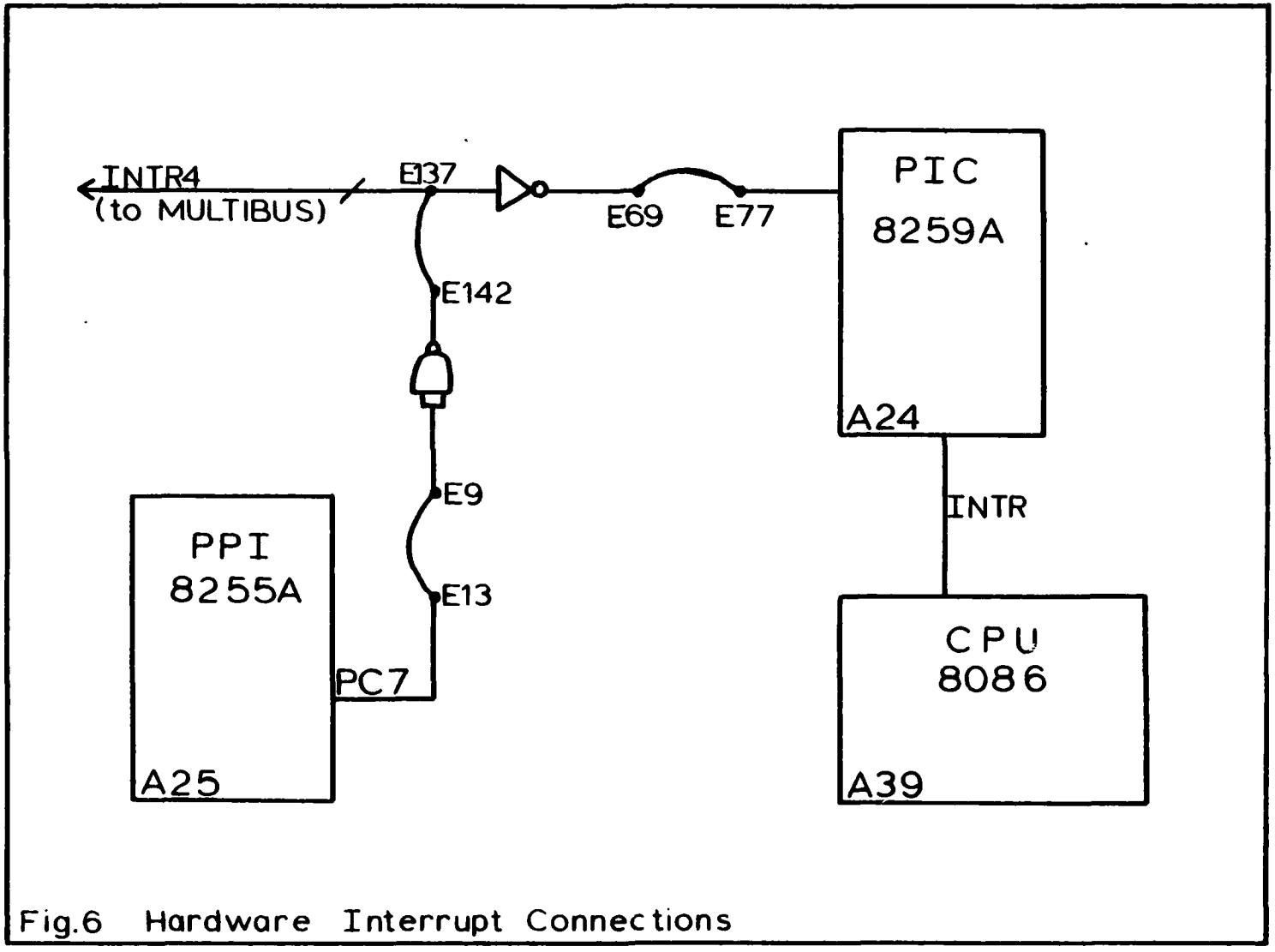


Fig.6 Hardware Interrupt Connections

IV. DETAILED SYSTEM DESIGN AND IMPLEMENTATION

A. STRUCTURE OF THE OPERATING SYSTEM

The distributed modules of the operating system create a virtual machine hierarchy which controls process interactions and manages real processor resources. The operating system is not aware of the details of process tasks. It knows each process only as an entry in the virtual process map. It provides processes with scheduling, synchronization, input/output services, and a diagnostic process.

The operating system is constructed in terms of four layers of abstraction. Each layer, or level, builds upon the resources created at lower levels (See Figure 7).

Level 0 is the bare machine which provides the physical resources, real processors and storage, upon which the virtual machine is constructed.

Level I is the scheduler and operating system support functions. This is the first layer of the software. It is closest to the hardware and encompasses the major machine dependent aspects of the system. Since several of its functions are directly involved with the hardware, it was necessary to code some of it in assembly language. This section also contains the "starting point of the operating

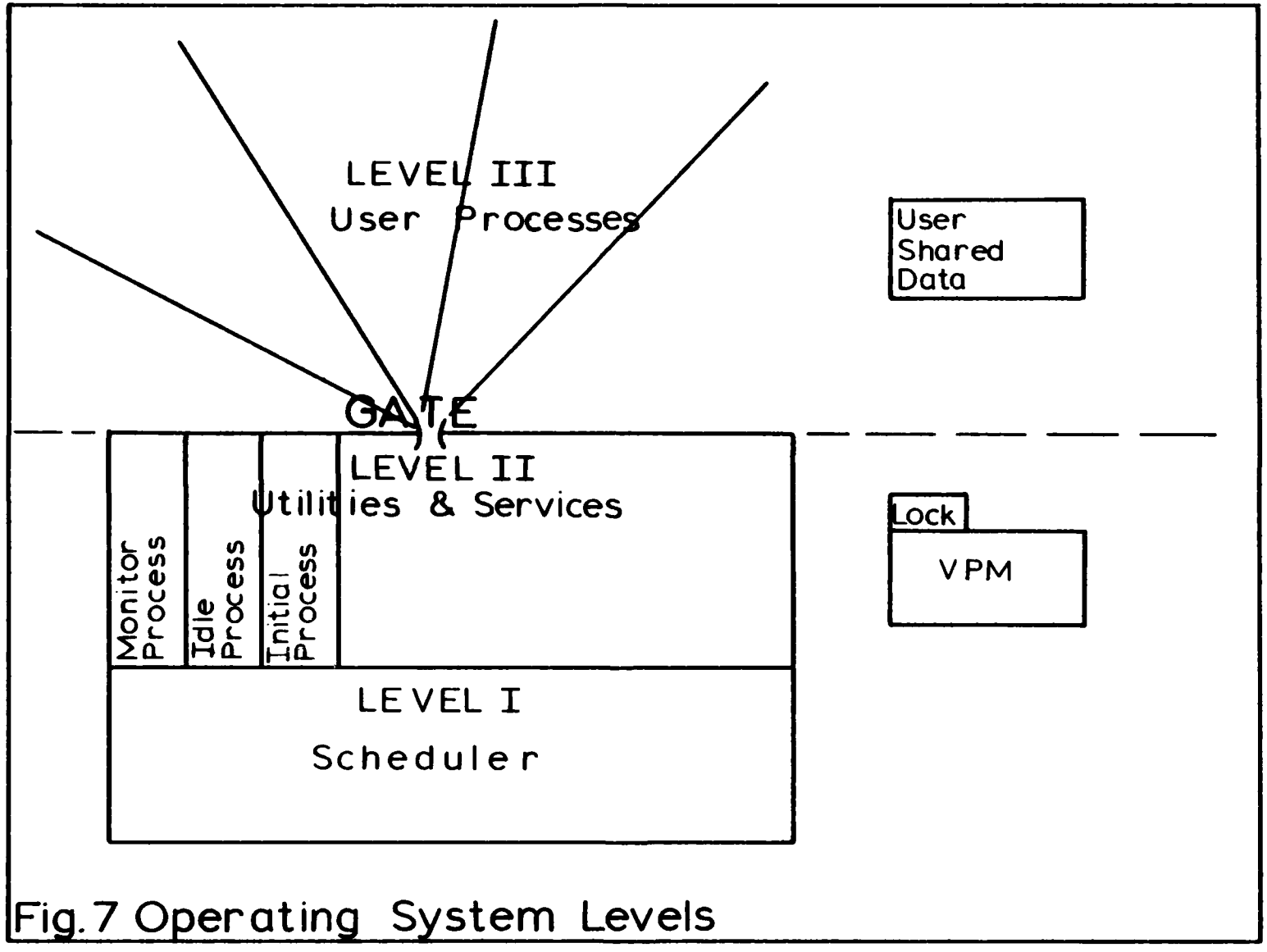


Fig. 7 Operating System Levels

system" and a small section of code to initialize physical devices.

Level II is the utilities/services level. All user available services are located here. Input functions were added and the "Preempt" function was corrected and changed to reflect the addition of a new system process.

Level III is the supervisor. It is essentially, the Gatekeeper module of Appendix I, and provides parameter adjustments for services requested by the user and a single entry point link to the operating system "Gatekeeper" in Level II.

Three system processes are "given" to the user. They are system processes as opposed to user processes because they are created by the operating system. The initial process, the idle process, and the new monitor process are scheduled by the same rules as any user process.

B. SYSTEM DATA BASE

A software lock is provided for the global data base used by the operating system. It is called "GLOBAL$LOCK". It has nothing to do with any shared data structures in global memory set up by user processes, but only the system data base, which is resident in shared global memory.

1. Virtual Processor Map

The Virtual Processor Map (VPM) is the major data base on which scheduling decisions are based. The table is partitioned into sets of virtual processors. There is one entry in the table for each virtual processor in the system and one virtual processor for each process if it has ever been created. Currently, the table is set to handle ten virtual processors for each real processor. In addition, the table can handle ten real processors. Three process entries occur by default for the idle, initial, and monitor processes. These processes are created and initialized by the operating system. A real processor will have ten contiguous entries in the VPM, seven of which are available to the user. As the VPM is an array of records, the index to each record or virtual processor structure is, in effect, a unique system ID for the virtual processor. Each entry consists of: a process ID given by the user, current state of the process, process priority, an eventcount thread, the threshold value for the event it is waiting for, and the stack segment base address for the process. The virtual process ID of FFH and FEH are reserved for the operating system. FFH is used by the idle process and the initial process. FEH is the unique identifier for the monitor process.

2. Eventcount Table

The eventcount table is for synchronization. It is an array set up for 100 possible entries consisting of an event name, current value, and an eventcount thread. Associated with each entry is an index. Each eventcount thread is the start of a blocked linked list. If the thread value is FFH, then that is the end of the list. Otherwise, is equal to some index value in the VPM. That VPM entry, or virtual process, will be active and waiting for a certain threshold value for that eventcount. The VPM entry may itself contain another thread value indicating that some other virtual process is waiting on the same event. However, it could be a different threshold value. Associated with this eventcount array is the variable "Events" which indicates how many active events there are currently in the table.

One event name, FEH, is reserved for the operating system. It is used to block processes that will never be awakened by the normal "Advance" mechanism. The initial process and monitor process are examples. See any INIT module in the appendicies.

3. Sequencer Table

The Sequencer Table is used to support process synchronization and ordering. Space is reserved for 100 sequencers. Each entry consists of a sequencer name and a value. Associated with the table is the variable

"SEQUENCERS" which indicates the number of active entries in the table.

4. Other System Data

Another structure in shared global memory is the Hardware Interrupt Flag array. There is one flag for every real processor, so the maximum size of the array is ten. The index of the array corresponds to the system unique CPU number for each real processor that is maintained in each local memory.

There are two utility variables kept in the system data base; (1)the number of active real processors, and (2) an array that keeps track of the number of virtual processors active on each real processor. The index of the array corresponds to the system unique CPU number for each real processor.

"CPU$INIT" is a byte variable that is accessed and incremented once each time a real processor becomes active. It is used to establish the system unique CPU number that is the key to the way all of the information in the system data base is organized. The reason this method is used, as opposed to the "Ticket" operation, is to preserve the hierarchy of the operating system.

C. LOCAL DATA BASE

The local data base for the operating system is the Processor Data Segment (PRDS). It contains information that applies only to the real processor on which it is resident. It consists of the system unique CPU number, the index of first entry in the VPM for that real processor, the index of the last possible entry in the VPM for that real processor, a counter that reflects the relative amount of time spent in the idle process, and a redundant variable that reflects the number of virtual processors currently assigned to the real processor. This number is maintained in shared memory and it is derivable from the second and third items in the PRDS.

D. INTERRUPT STRUCTURE

The operating system controls the multiprocessor environment. A means of communication or signaling between real processors is required. A single, system-wide preemptive interrupt signal is the vehicle to accomplish this. It was arbitrarily chosen to be interrupt-request-4 of the eight available for use. The hardware connections required to implement the system interrupt are described in detail in Chapter III, Section D. Software control is required to ensure that the hardware performs with certain characteristics and to actually issue the interrupt. Several problems were encountered in the testing of this part of the system.

A brief description of the entire interrupt sequence is in order first. The operating system finds, in the course of events, that one or more processes are waiting for an event that the currently running process just caused. If any of those waiting processes are associated with the same real processor, then the scheduler will simply be called to make a scheduling decision. If those waiting processes are associated with other real processors, then the operating system has three things to do: (1) It locates the waiting processes in the VPM and changes their state to "ready", (2) It sets the hardware interrupt flag associated with each real processor, and (3) it issues the single system wide interrupt signal to indicate to all processors that a scheduling may be in order. All processors will execute their interrupt handler which in turn will check to see if their respective flags are set. If the flag is set, a call to the scheduler is made. If it isn't, normal execution resumes. The adopted convention in this implementation is that the processor issuing the interrupt never sets its own flag. If it needs rescheduling, the scheduler is called directly.

At this point it would be good to review Figure 2 in Chapter III, Section D. Note the positions of the interrupt driver and the other inverter. There is also a pull-up resistor on the INTR4 line on the MULTIBUS side of the interrupt driver. More complete coverage of the hardware

devices and their programming is given in [Ref. 3:pp.B-106 to B-123] and [Ref. 3:pp.A-136 to A-136] for the PIC. The best reference for the PPI is [Ref. 2].

All of the initialization software for the PPI and PIC is correct. It will be explained at the end of this section. However, the misunderstanding of one of those programmed features caused the poor design of a very small but critical portion of the operating system. There are two procedures that issue interrupts, "Preempt" and "Advance". Cox arranged the instructions for issuance of the interrupt in the following manner:

```
OUTPUT(PORT$CC) = RESET;
OUTPUT(PORT$CC) = 80H;
```

In one procedure interrupts were disabled during this sequence and in the other they were not. Now they are both disabled consistently. The above sequence is not logical because the INTR4 line on the MULTIBUS is not reset after being used. Thus, no one else can use it. Other processors would not have their interrupts recognized unless the original processor that sent an interrupt, reset it. They could send it, but it would have no effect on a MULTIBUS interrupt line already being held low. That could have disasterous consequences. Cox's demonstration of two processors with three processes worked by accident. The sequence was the only way he could get the other processors to recognize an interrupt signal. Supposedly both

processors were issuing interrupts, when in fact only one was. The demonstration appeared to work because of timing. An interrupt signal causes all processors to check the hardware interrupt flags. Therefore, even if an interrupt signal was missed, the hardware interrupt flag was at least set. If the one processor that had control of the interrupt request line could issue another interrupt, then rescheduling would occur none the less. Additionally, the controlling process in Cox's demonstration was ten times as long as the other two combined on the other processor. This combination of features allowed the system to synchronize. The moment additional SBC's were added to the system or the system was loaded down with more processes, it failed to synchronize and the entire system would idle.

The first step in solving the interrupt problem was to reverse the two instructions because that was a requirement for the system to operate asynchronously. Diagnostic hardware was used to determine why no interrupts were recognized from this "correct" sequence. It was found that although the PIC was programmed to be edge triggered, the active level had to be maintained for a short period before the interrupt request was recognized by the CPU. In the corrected sequence, the interrupt request was apparently being removed or reset too quickly. After arbitrarily introducing a delay, all other CPU's recognized the request. It is suggested that an interrupt acknowledge

system of flags be set up so the interrupting CPU will have a positive indication that interrupted processors reacted to the signal.

In testing the generality of MCORTEX once the interrupt problem was tentatively cleared up, additional SBC's were added to the system. It was found that an uninitialized SBC would hold down the INTR4 line. This will kill all communications between processors. If the additional SBC's were loaded with MCORTEX and no user processes, then the system communicated effectively because the operating system would initialize the PIC's and PPI's. This is the reason in Appendix A that any SBC with no user process, "idle CPU's", are initialized first.

MCORTEX has now been tested with four SBC's. One SBC was maintained as an "idle CPU". Three were loaded with five processes. The original three processes of Cox with slight changes were kept. Then an independent system of two processes was added. The two independent systems share one of the SBC's. In addition, the new system actually demonstrated the sharing and passing of information via common memory. Prior demonstrations only tested syncrhonization. To prevent the system from scheduling processes in a repeated sequence, input was prompted from a user. This new demonstration showed truly asynchronous processes working together and interfering with each other.

However, this demonstration does not guarantee the correct operation of the preemptive interrupt.

A final point needs to be emphasized at this point. The question has been asked, "Will interrupt requests from different processors ever overlap?" The answer is no. They are guaranteed mutual exclusion because the system data structures are locked before the interrupt request is issued. No other single board computer can issue an interrupt until it controls the lock.

To implement the interrupt structure, the PIC and PPI have to be initialized. The triggering mode, interrupt vector table address, and the parallel port mode have to be set. The type (number) of interrupt received is added to the interrupt vector table control word and multiplied by 4. The result points to a 2-word (4-byte) memory space in the interrupt vector table -- called the interrupt vector. The interrupt vector points to the address of the intended interrupt handling procedure. The interrupt vector consists of the offset and code segment base address for the interrupt handler. An interrupt will cause the flag register, CS register, and the IP register to be pushed onto the stack. Then, an indirect call is made through the interrupt vector to the interrupt handler.

Initialization of the 8259 PIC must follow a specific format and sequence. It can require up to four interrupt

control words (IWC) for initialization. In our case, only three are required: IWC1, IWC2, and IWC4.

IWC1 must go to port 0C0H. It is 13H. It means edge triggering, no slave PIC's, and IWC3 is not required.

IWC2 must go to port 0C2H. It is 40H. It indicates that the interrupt vector table starts at 40H x 4 which is 100H. IWC4 must go to port 0C2H. It is 0FH. It indicates 8086 mode, automatic end of interrupt, and buffered mode.

The result of the initialization is that in the case of an INTR4, the PIC will perform the following operation:

4x(IWC2+INTR#) = 4x(40H+4) = 110H

The result is that the interrupt vector address for an INTR4 is 110H. The two words residing at 110H and 112H will be used for an indirect call to the INTR4 interrupt handler:

[110H]:[112H]

The starting address of the interrupt handler must be loaded to this vector address.

There are three parallel I/O ports. One bit of one of the ports is used to drive the INTR4 line. Port C and bit 7 are used. See Figure 2 for details on connections. One control word is sent to port 0CEH to indicate that port C will be used as an output port. The control word is 80H.

E. PROCESS STACK

The per-process stack is used to form a virtual processor which maintains the process state information. This includes the current execution point, the code and data segment base addresses, and in a particular instance, all the working registers. Since this stack reflects the state of the real processor registers at the time of execution, this data structure represents a virtualization of the real processor. By loading the stack segment base location into the SS and restoring the real processor status from the stack contents, the process execution can be continued where it last left off. To change processes, the real processor status is saved on the current process' stack for future use and the new process' stack segment is loaded into SS from the VPM. The real processor status is then restored *from* this new stack segment.

A three word header is maintained at the stack segment base: stack pointer (SP), base pointer (BP), and interrupt-return-type (INTR$RET). This header is used to make the proper adjustments and decisions for restoring the hardware condition during scheduling.

Each process must have its own unique stack. The per-process-stack is automatically blocked out approximately 125 words of user run-time stack.

F. SCHEDULING

The scheduler has two different entry points and two different exit points. The difference exists because of the different requirements for saving registers.

A normal entry call by a Level II function does not require any general registers to be saved. The only register which must be explicitly saved on the stack is the data segment register (DS). At this time, INTR$RET is set to "0", indicating a normal scheduler call. The scheduler executes and process selection and initiation take place.

An interrupt entry call is made through the interrupt handler which checks the hardware interrupt flags to see if it is the one for which the interrupt is intended. If so, all registers are saved in a particular order. INTR$RET is set to 77H, indicating an interrupt scheduler call. As DS was already saved, the scheduler is entered at a different point to avoid that instruction. Process selection and initiation then take place.

In the common code, the past process' stack status is saved in the stack heading. Also, the return type indicator is saved in the stack header to be used when the process is ever rescheduled. "Getwork" is called to select the highest priority process that is ready to run. The stack segment for the selected process is returned in AX. "Getwork" received the information from the VPM. The scheduler loads AX into the stack segment register. The context has

effectively been switched. The new process' stack condition is restored and the return type indicator is checked to ascertain the circumstances of the process' last run-time termination. If the indication is a normal return, DS is restored and the process returns to its calling point. If the indication is an interrupt return, then all registers are restored and the process returns to its execution point at the time it was interrupted.

The scheduler makes calls to two other utility functions: "RDYTHISVP" and "RET$VP". "RDYTHISVP" determines which process is currently running by calling "RET$VP" and sets it to ready.

G. THE GATE AND GATEKEEPER

The "Gate" module provides parameter translation and adjustment to facilitate processing. It directly provides the public link with user processes for access to system services. A call to the specific utility function is processed in the "Gate", from which a further call is made to the operating system "Gatekeeper",where a final call is made to the specific procedure processing the requested service. The "Gate" module must have been provided the "Gatekeeper" procedure's address before the "Gate" was compiled. This address is obtained manually from the operating system location map in file: "KORE.MP2" (see end of App. F).

H. USER AVAILABLE SERVICES

1. Create Eventcount

A process cannot assume that an eventcount it is going to use is already in existence. Any process that will use the eventcount must create it. The procedure is passed one parameter: the name of the event to be created in the eventcount table. It calls "Locate Eventcount" to see if the named event already exists. If not, it enters the name in the table and initializes the entry. It then increments the number of events; "EVENTS". If the named event is found when "Locate Eventcount" is called, it simply returns with no change.

2. Advance Eventcount

The "ADVANCE" procedure increments the named eventcount and broadcasts this to the processes awaiting this particular event, via a thread. (See Chapter IV, Section C, Subsection 2 for an explanation of the thread.) To awaken a process, its state is set to ready. If the awakened process is associated with the same real processor, then the scheduler is called to reschedule all eligible processes (ready processes) for the real processor. If the awakened process is associated with another real processor, then the hardware interrupt flag for that real processor is set and an interrupt is issued.

3. Await Eventcount

"Await" is passed two arguments; (1) the event's name, and (2) the threshold value of the event it is waiting on. The current value of the named eventcount is checked. If the current value is less than the threshold value, the process is blocked and its state is set to waiting. The blocked process is added to the head of the thread. The scheduler is then called to select another eligible process to run. If the process is not blocked, "Await" executes a return.

4. Read Eventcount

"Read" is passed two parameter; (1) the eventcount name, and (2) a return pointer. The procedure obtains the current eventcount value and returns it to the calling process by using the pointer as a base for the value to be returned. With indirect calls, a value can not be returned directly.

5. Create Sequencer

"Create Seq" is passed the name of the sequencer to be created. The Sequencer Table is searched by "locate Seq" like the Eventcount Table to see if it exists. If it exists, then a simple return is executed. If it does not exist, the name is entered into the table and the other entries are initialized. The number of entries in the table, "Sequencers", is incremented by one. As with

eventcounts, all processes that use a sequencer must create it first. They can not assume it has already been created.

6. Ticket Sequencer

The "Ticket" procedure is passed two parameters; (1) the name of an already created sequencer and (2) a return pointer. Its purpose is to obtain a unique sequential number for the named sequencer. The current value is returned via the base pointer. The sequencer is then incremented. With indirect calls, a value can not be returned directly.

7. Create Process

"Create Proc" is called from the initial process or one of the user processes. The parameter passed as input to the "Gatekeeper" is a pointer. The pointer is used as a base to a structure to overlay the data parameters supplied by the user. These user supplied parameters are structured in the "Gate". The parameters are: process ID, process priority, process stack segment location, process IP, and process CS. CS:IP is the starting address of the user process and it is manually obtained from the memory map associated with the process (the map may apply to several processes). Once obtained it is inserted into the initial module that created it. That module must then be recompiled, relinked, and relocated. A per-process stack is established and initialized. The PRDS table and number of virtual processors on this real processor is updated.

8. Preempt Process

"Preempt" is passed only the name of the process to be preempted. It searches the VPM for the process ID. When found, the process is set to ready and the scheduler is called or the processor is interrupted depending on whether the preempted process is the same real processor or not. "Preempt" is intended for high priority processes that block themselves when finished.

"Preempt" was chosen as the vehicle to implement the second goal of this thesis. Thus, the existing frame work of the operating system was used to implement the "Monitor Process". The procedure has a second part that is executed if the process name is "FEH". That name has been reserved for the "Monitor Process" which is described in the next section. The "Monitor Process" is a high priority diagnostic tool given to the user. Like the other system processes, the "Idle Process" and the "Initial Process", the process is associated with every real processor. "Preempt" has to search each real processor's set of entries in the VPM to find each "Monitor Process" and set each one to ready. It must set the hardware interrupt flags of the other real processors, but not its own. It issues an interrupt and all the other real processors are forced into the highest priority process, the "Monitor Process". For itself, the scheduler is called and it goes into the monitor process. With the input services that have been added to

MCORTEX, the monitor can be scheduled via "Preempt" on a regular basis or by request.

9. Communications with Console

ASCII output is by single character or line. "Out Char" is given a single byte value and "Out Line" is given a pointer to the beginning of a string. The string should end in "%" which will stop output. Any ASCII output can be stopped at the CRT by a "^S" which also freezes the process. ASCII output is resumed with a "^Q". This ability is useful to freeze diagnostic output for study.

ASCII input is by single character and it is not echoed. "In Char" could easily be used to fill a buffer for line input. Once "In Char" has been invoked, the procedure waits for the character. An additional service should probably be added that does not wait.

Hexadecimal input and output exists for byte and word values in: "Out Num", "Out Dnum", "In Num", and "In Dnum". Output is interruptable. Input is echoed for both byte and word values. Illegal characters are ignored.

I. SYSTEM INITIALIZATION AND SYSTEM PROCESSES

1. System Initialization

The starting address for initiating the operating system is found in the operating system memory map in file: "KORE.MP2" (See App. F). It is usually 100:30. When the system is initiated, by issuing the monitor command,

.G100:30<cr>, the operating system initialization routine is executed. The PPI and PIC are initialized as discussed in Chapter IV, Section E.

Next a unique, sequential number is obtained and used to initialize the "CPU$NUMBER in the PRDS table for this processor. This is the identity of this real processor and is the key to the location of entries in the VPM. The PRDS is further initialized by using the "CPU$NUMBER" to calculate the beginning and ending locations in the VPM for this processor and by setting the number of virtual processors on this real processor to three.

The VPM is then initialized for the three system processes. The number of real processors in the system, "NR$RPS", is updated to indicate another processor has been added to the system. ("CPU$INIT" could have been used.) All hardware interrupt flags are cleared and the scheduler is called.

The stack used during the initialization is the kernel stack, which is located at the location specified in the execution of "LOC86" (see App. F). In this case, the kernel stack is at 3000H and its size is 75H. In order to initialize the stack segment, SS, and stack pointer, SP, the initialization routine code cannot be in a procedure block. It must be coded as the main routine operating system ("KORE") module. Neither the initialization routine or the kernel stack is used again during the system run time.

2. Process Initialization

When the operating system initialization routine is completed, it calls the scheduler. The scheduler selects the highest priority ready process to run. Initially, that will always be the initial process. This process is intended to be the vehicle by which the user processes are established in the tables of the operating system. It is assumed that the programs are already loaded in memory as characterized in the creation specification.

During the linking and locating of the operating system, the file, "INITK", was included and located at 2800H. This file acts as a dummy to establish the space and starting location for the operating system's reference. It is a valid initial process module, but it will normally be overlayed by an "INIT" procedure modified and loaded by the user. The user must locate his "INIT" module at 2800H in this implementation.

The "INIT" file must fit the prescribed format as provided and enough space must be reserved for it by the locate command. The user only modifies one area of the file. He provides the absolute parameters required in the call to the "Create Proc" procedure. One call must be made for each process. The user can create up to 7 processes per real processor for a system total of 70.

Also, of concern to the user, is the intended stack location. At the desired stack location, the user must

allow enough space for a stack of 120H bytes. The user process must be coded in procedure blocks so the stack segment, SS, and the stack point, SP, will not override what is provided by the operating system. The user process procedure block should have the attribute "public" so that the address can be obtained from the location map to be used as parameters for process creation. See Section I, Subsection 7 of the chapter.

The "INIT" process will create the processes specified, then block itself with a call to "Await". This allows the newly created processes to be scheduled and run according to their priorities.

3. The Idle Process

The idle process has the lowest priority of any process. It is selected by the scheduler only if there are no other eligible processes (i.e., all others are blocked). When selected, its only function is to update the counter contained in the real processors own PRDS table at approximately one second intervals. Thus, a rough measure of time is obtained when this real processor is doing no useful work.

4. The Monitor Process

This high priority system process is scheduled by all processors at the same time. No matter which process preempts the "Monitor Process", the entire system will be "put to sleep". The entire system address space is then

accessible from the serial port of any processor simultaneously. Each processor exits the "Monitor Process" individually and in any order. The user could leave an "idle CPU" in the monitor. However, the user must realize that data could be changing while he is accessing it. The system would then resume normal scheduling.

A primary motivation for implementing the "Monitor Process" was that there was no way to examine memory for diagnostic purposes until after the system had run its course or the designer stopped it arbitrarily. Now, memory can be examined on a synchronized basis with the occurence of specified events. A record of transactions with the process can also be secured to a line printer simultaneously.

Appendix C contains a summary of the monitor commands which closely mimic those of the resident monitor.

J. METHODS AND FACILITIES

Software development for MCORTEX was accomplished on an Intellec MDS 800 developmental system under the INTEL Systems Implementation Supervisor (ISIS-II). The MDS 800 is based on the 8080 microprocessor. ISIS-II is a diskette operating system. The MDS system has two double density disk drives. Object code for the 8086 is developed on the MDS 800 and down loaded to the iSBC86/12A's. Readers unfamiliar with the ISIS-II system are refered to [Ref. 3].

However, all source code was developed using the text editor, TED, from Digital Research Inc.

There are two double density eight inch diskettes necessary to continue implementation of or to examine MCORTEX and test process source code. KLINEF.A1 contains module source code, relocatable code modules, executable code modules, memory maps and basic ISIS-II utilities. KLINEF.B1 contains all the module processing programs such as the linker, locator, assembler, and compiler. Because, these programs are large, they are stored on a separate diskette. Output from the module processing programs goes to KLINEF.A1. Appendix C contains annotated directory listings of both diskettes.

External hardware connections required to set up the MDS 800 and SBC's in order to facilitate communication are detailed in Appendices A and B. Appendix A is a detailed "pre-power-on" and "post-power-on" checklist to load the single board computers. Appendix B contains a drawing that describes the physical make-up of the transfer hardware. Cox cites a five wire RS232 cable in [Ref. 12:p.45]. However, only three are required. See Appendix B for further details.

1. The PL/M-86 Compiler

As software modules must be compiled individually, no command files for use with the ISIS-II utility, SUBMIT, were established [Ref. 3:pp.3-13 to 3-14]. The following

controls were found to be useful: PRINT(:LP:), NOPRINT, CODE, and LARGE. The use of the control LARGE is mandatory as discused earlier. It causes the compiler to represent addresses in such a way that the whole megabyte range of the 8086 can be used. See Chapter III, Section C, Subsection 2 for additional information on addressing modes. The following references contain more details concerning run-time representations by the compiler when the LARGE control is used: [Ref. 7:pp.3-13 to 3-14, 5-1 to 5-5, 8-1]. Other pertinent references concerning the compiler are: assembly language module linkage [Ref. 7:pp.9-1 to 9-3] and the preemptive interrupt process [Ref. 7:pp.10-1 to 10-4].

2. Link86(Linker)

LNK86 takes object code modules, combines them, and resolves external references from each individual module. The resulting relocatable file has the default file extension, ".LNK". No controls are necessary. Two command files, "LNKK.CSD" and "LNKP.CSD", which have been established can continue to be used in the future. "LNKK.CSD" contains the commands to link all the object code modules that compromise MCORTEX. "LNKP.CSD" contains the three commands that link the user modules into the three modules that will, after further processing, be loaded onto three SBC's. Complete error listings are given in [Ref. 4:App. A].

3. Loc86(Locator)

Use of the locator is a little more involved. This program assigns addresses to the relocatable code modules that come from the linker. It requires knowledge of how modules are organized in the system address space both by the user, and by the operating system designer. [Ref. 4:Chapter 4] gives the best description of how the locator handles modules. The two command files, "LOCK.CSD" and "LOCP.CSD", established for the locator will also provide most of the knowledge required to locate user code. Eventually, after more extensive testing, MCORTEX will be tuned and compressed to the maximum extent possible so that it can be put on EPROM. Currently, modules are well dispersed to facilitate design and development.

One type of output file from LOC86 is the ".MP2" memory map file. In addition to valuable diagnostic information, several critical pieces of information from these file are necessary for the correct operation of MCORTEX and the user process. There are essentially three items; (1) the location of the gatekeeper, (2) the starting addresses of user code, and (3) the starting point of the operating system. Warning remarks in the source code detail what and where these items are. In addition, later sections in this chapter will discuss those items.

Upon examining any ".MP2" file, a "WARNING 56:..." will at certain times appear. For example, see KORE.MP2 in

Appendix E. This is caused by intensionally overlaying some kind of segment in space that was previously reserved. The warning should not cause too much concern. See the memory map from P03 in Appendix K for a more clear cut example. The effort in the last example was to prevent any code being put next to the initial process code. See [Ref. 4:Chap. 3] for more details on the controls available to the LOC86 command.

V. CONCLUSIONS

The pricipal goals of this thesis were met. The generality of MCORTEX was tested and demonstrated. Two problems were uncovered: (1) The interrupt mechanism does not appear to be totaly satisfactory and (2) the issue of an inactive SBC bringing down the system will eventual have to be addressed. The system was demonstrated with four single board computers and there appears to be no reason why the additional six cannot be added. Actual synchronized sharing of data was demonstrated. Two independent user systems operating simultaneously was demonstrated. The system was prevented from scheduling processes in a fixed sequence by introducing user interaction. User input services have been added to MCORTEX and a method was found within the existing framework of the operating system to incorporate dynamic interaction with the operating system itself. The total address space is now accessible from any single board computer for system debugging and examination. Additionally, the operating system can be continued without reinitializing the system. "Preempt", which had never been tested, is now used to evoke the "Monitor Process".

Three possible problem areas were cited by Cox in his thesis. The cause of the first one was acertained. A weakness in the interrupt system has been clearly

identified. A possible problem with the eventcount thread was not examined as it did not immediately impact on the primary goals of the thesis. The third problem cited was the inablitiy to stop and restart the system without reinitializing all code. That ability now exists by selectively preempting the "Monitor Process".

Future research with MCORTEX should concentrate on taking precise timing and performance measurements while the system is heavily loaded down. The second possible problem cited by Cox, mentioned above will also have to be examined. The process stacks should also be examined to dynamically confirm capacity usage as there are no definitive guidelines upon which to base stack size. A system of hardware interrupt ackowledge flags will probably have to be set up to stregthen the preemptive interrupt system.

APPENDIX A

SYSTEM INITIALIZATION CHECKLIST

I. PRE-POWER-ON CHECKS

A. SBC's have address translation switches and jumpers correctly set.

B. SBC's have 3 interrupt jumpers set.

C. SBC #1 has MULTIBUS clock jumper set.

D. No other SBC has MULTIBUS clock jumper set.

E. SBC's and common memory board full seated in odd slots of MULTIBUS frame.(RAM board can be in any slot.)

F. Remove all serial CRT cables from SBC's.

G. J2 26 pin edge connector on transfer cable can be hooked up to one of the SBC serial ports at this point.

H. If RS232 transfer-cable has a "null modem" switch on it, set it to "null modem". This transposes wires 2 and 3. The switch may alternately be marked "computer to computer" and "computer to terminal". Set to "computer to computer". It should always remain in this position.

I. Connect other end of transfer cable (25P RS232 connector) to 2400 baud CRT port of the MDS system.

J. Connect any CRT to the 9600 baud TTY port of MDS system.

K. Ensure CRT is set to 9600 baud.

L. CRT that will be connected to SBC's should be set to 9600 baud. This step is not mandatory, but recommended.

M. Each CRT that will be connected to SBC should have RS232 cable hooked up to serial port. It should lead to flat 25 wire ribbon and J2 connector so it can eventually be hooked to serial port of the SBC's.

II. POWER ON PROCEDURES

A. Turn power-on key to ON at MULTIBUS frame.

B. Press RESET near power-on key.

C. Turn power on to all CRT's.

D. Power up MDS disk drive.

E. Power up MDS terminal (If not already done).

E. Turn power-on key to ON at MDS CPU (front panel, upper left corner).

F. Line printer can be turned on at any time.

III. BOOT UP MDS

A. Place system diskette in drive 0. Executable modules and SBC861 can be on another diskette in drive 1.

B. Push upper part of boot switch in (It will remain in that position).

C. Press reset and release.

D. When interrupt light #2 lights on front panel, press space bar on console device.

E. Reset the boot switch by pushing lower part of switch.

F. ISIS-II will respond with "-".

G. Line printer can be turned on at any time.

IV. LOAD MCORTEX AND PROCESS MODULES

A. Type "SBC861<cr>".

B. If "*CONTROL*" appears, SBC was not able to set its baud rate. Press RESET on MULTIBUS frame and start over. Once set, all SBC's should accept modules.

C. If "Bad EMDS connection" appears, you will not be able to continue. Check connections. Make sure diskette is not write protected. Push RESET at frame. Try again.

D. SBC861 will prompt with ".". It will now accept any monitor command.

E. Type "L KORE". Wait for ".".

F. Type "L <process filename>". Wait for ".".

G. Type "E" to exit SBC861. It is not a good policy to switch the transfer cable to another SBC serial port with out exiting SBC861.

H. Switch transfer cable to next SBC. Go to A.

V. RUN MULTIPROCESSOR SYSTEM

A. Disconnect transfer cable from last SBC loaded.

B. Connect J2 connector from each CRT to an SBC serial port.

C. After all CRT's connected, push RESET on frame to brake baud rate.

D. On each CRT press "U" to obtain monitor. Will respond with ".".

E. Type "G100:30<cr>" at each terminal to start

MCORTEX. If one of the SBC's is running only MCORTEX and no user processes, start it first.

APPENDIX B

SYSTEM INITIALIZATION HARDWARE CONNECTIONS

The following page contains a drawing of the hardware connections required for transfering code developed on the MDS 800 to the single board computers. The transfer cable has two parts, an RS232 cable and a 25 wire ribbon.

The RS232 cable has a 25P connector that mates with the 2400 baud 25S connector on the back panel of the MDS CPU. The other end is a 25S connector that will mate with the 25 wire ribbon.

The ribbon has a 25S connector for attachment to the cable. The other end is a 26 pin edge connector that will mate with the J2 junction on the single board computer. J2 is the serial port. The 26th pin is left unconnected on the edge connector.

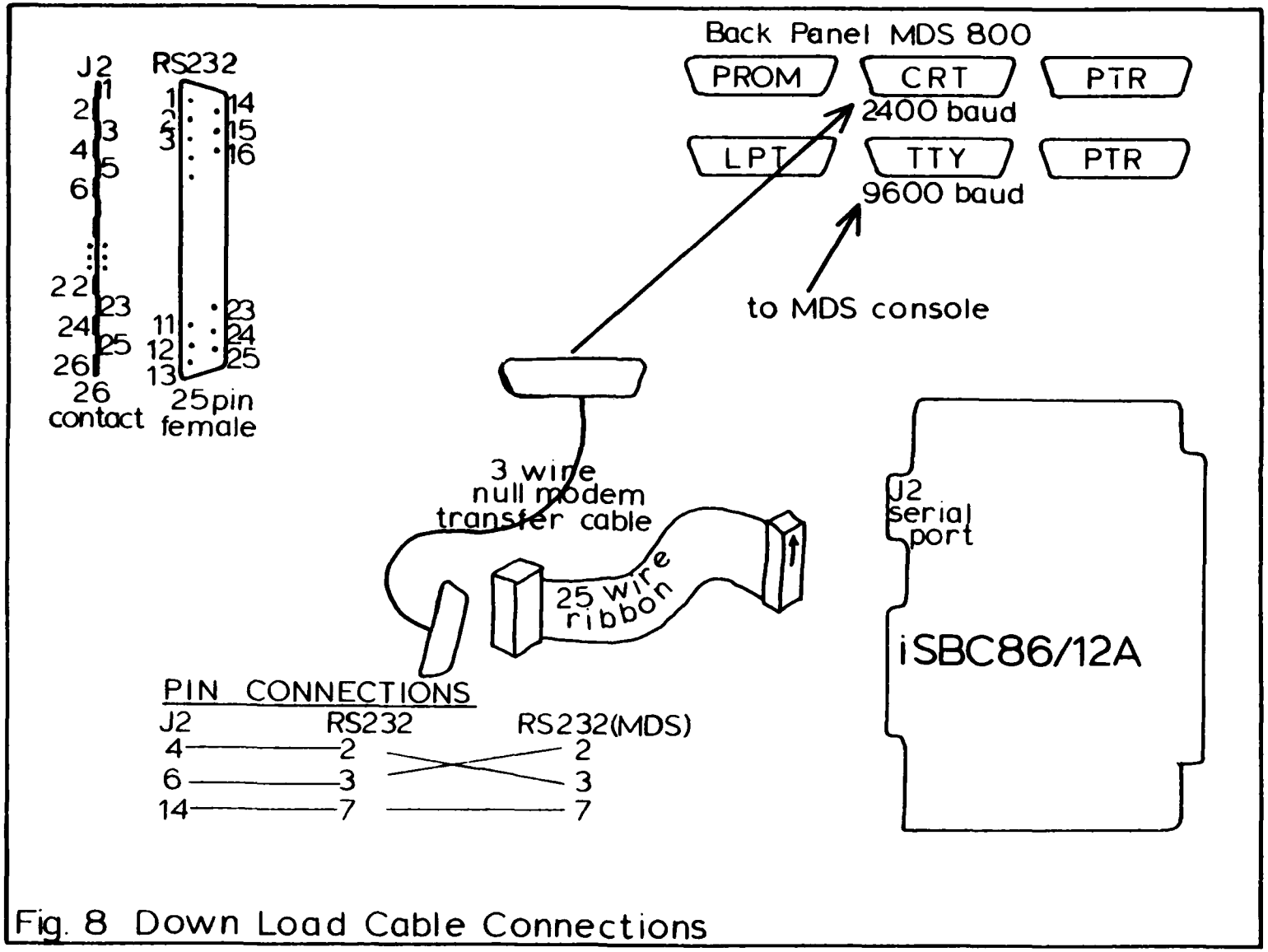


Fig. 8 Down Load Cable Connections

APPENDIX C

ANNOTATED DIRECTORY LISTING FOR KLINEF.A1

NAME	.EXT	REMARKS
ATTRIB		ISIS-II Utility. See Ref. 3
COPY		" " " "
DELETE		" " " "
DIR		" " " "
FIXMAP		" " " "
HDCOPY		" " " "
IDISK		" " " "
LIB		" " " "
LIB86		" " " "
RENAME		" " " "
SUBMIT		" " " "
TED		Text editor. Not a ISIS-II utiltiy.
LNK	.CSD	Links MCORTEX modules. Command file.
LNKP	.CSD	Links process modules. Command file.
LOCK	.CSD	Locates MCORTEX relocatable file. Cmd file.
LOCP	.CSD	Locates process relocatable files. Cmd file.
GATE	.SRC	User gate module source code.
GATE	.OBJ	" " " object " .
GLOBAL	.SRC	MCORTEX global data base source code module.
GLOBAL	.OBJ	" " " " object " " .
INIT1	.SRC	User initial process source file for 1st CPU.
INIT2	.SRC	" " " " " for 2nd CPU.
INIT3	.SRC	" " " " " for 3rd CPU.
INIT1	.OBJ	Compiler object code output file for 1st CPU.
INIT2	.OBJ	" " " " " " 2nd CPU.
INIT3	.OBJ	" " " " " " 3rd CPU.
INITK	.SRC	MCORTEX initial process source code module.
INITK	.OBJ	MCORTEX initial process object code module.
KORE	.LNK	MCORTEX relocatable code file.
KORE		MCORTEX executable code module.
KORE	.MP1	Linker map file.
KORE	.MP2	Locator map file.
LEVEL1	.SRC	MCORTEX level1 source code module.
LEVEL1	.OBJ	" " object " " .
LEVEL2	.SRC	" level2 source " " .
LEVEL2	.OBJ	" " object " " .
PROC1	.SRC	User process 1 source code.
PROC1	.OBJ	" " " object " .
PROC2	.SRC	" " 2 source " .
PROC2	.OBJ	" " " object " .
PROC3	.SRC	" " 3 source " .
PROC3	.OBJ	" " " object " .
PROC4	.SRC	" " 4 source " .
PROC4	.OBJ	" " " object " .

```
PROC5 .SRC     "      "     5 source  "   .
PROC5 .OBJ     "      "     " object  "   .
P01   .LNK   User relocatable code file for CPU 1.
P01           "   executable     "   "     "   "   ".
P02   .LNK    "   relocatable    "   "     "   "   2.
P02           "   executable     "   "     "   "   ".
P03   .LNK    "   relocatable    "   "     "   "   3.
P03           "   executable     "   "     "   "   ".
P01   .MP1   Linker map file.
P02   .MP1     "      "     " .
P03   .MP1     "      "     " .
P01   .MP2   Locator "      " .
P02   .MP2     "      "     " .
P03   .MP2     "      "     " .
SBC861       SBC down load program.
SCHED .ASM   Scheduler & interrupt handler assembly
             language source code module.
SCHED .OBJ   Scheduler & interrupt handler object
             code module.
```

ANNOTATED DIRECTORY LISTING FOR KLINEF.B1

NAME	.EXT	REMARKS
COPY		ISIS-II utility.
DELETE		" " .
DIR		" " .
SBC861		MDS to iSBC86/12A down load program.
SBCIOC	.LIB	
SBCIOS	.LIB	
SBCIOL	.LIB	
SB957A	.020	
ASM86		ASM86 assembler.
ASM86	.OV0	
ASM86	.OV1	
ASM86	.OV2	
PLM86		PL/M-86 compiler.
PLM86	.OV0	
PLM86	.OV1	
PLM86	.OV2	
PLM86	.OV3	
PLM86	.OV4	
PLM86	.OV5	
PLM86	.OV6	
PLM86	.LIB	
LINK86		Object code linker.
LINK86	.OV0	
LOC86		Relocatable code locater.

APPENDIX D

SBC861 & The MCORTEX MONITOR

No up to date information on the down load program and the monitor for the SBC's was found. There is related material in [Ref. 1] and [Ref. 3:pp.B-179 to B-183] only. The "E" (exit) and "L" (load) commands are discussed in Appendix A. The remaining commands are the same as if you were dealing with the SBC monitor. Only the form using the two dimensional address is shown below. Only significant digits need be displayed.

Command	Meaning
.dzzzz:zzzz<cr>	Display contents at zzzz:zzzz.
.dzzzz:zzzz#zzzz<cr>	Displays contents starting at zzzz:zzzz for zzzzH bytes.
.dxzzzz:zzzz<cr>	Same rules as above,except byte are decoded in to basic assembly language.
.gxxxx:xxxx<cr>	Loads CS and IP with xxxx:xxxx and starts execution.
.x<cr>	Displays current register contents.
.c[name]<cr>	Change register contents.
.szzzz:zzzz	Subsititute at zzzz:zzzz. If followed by ",", next byte is displayed for possible substitution. <cr> causes prompt for byte value. "," or <cr> causes change to take effect with above rules still applying.

The MCORTEX MONITOR

The MCORTEX monitor closely parallels the command structure of the SBC monitor. There are only three commands; display, substitute, and exit. Commands are evaluated on a character by character basis instead of waiting until a buffer is filled. Once you are in the monitor, illegal characters are not accepted and pressing the wrong key will have no effect. If you enter the wrong command or address, follow through to start over. The full two-dimensional 8 hex character address must be specified. key. All commands are prompted for by a ".". See Chapter IV.

Command	Meaning
.d0000:0000<cr>	Display one byte at 0000:0000. ":" is automatically inserted.
#FF	Display FFH bytes starting at 0000:0000. FFH is maximum. Once last digit entered, formated display starts.
.s0000:0000_	Spa:e cause current contents to be displayed as: "xx-".
,	Casues offset to be incremented and new address deisplayed. Same rules apply.
<cr>	Returns monitor command prompt. After a space and the contents desired are displayed as: "xx-". the same or a new byte value must be entered.
00	A "," will continue the sequence to the next location. A <cr> will terminate it and return the prompt.
.e	Return the SBC to MCORTEX. This can be done in ANY order or not all for a particular terminal.

APPENDIX E

LEVEL II -- MCORTEX SOURCE CODE

All of the source code in LEVEL II is contained in file: LEVEL2.SRC. It is compiled with the LARGE attribute. It is one of the relocatable code modules in file: KORE.LNK. It is part of the executable code module in file: KORE. A memory map for this module is located at the end of Appendix F. All operating system calls available to the user are located in this module.

```
/**************************************************************/
/**************************************************************/
/*0009*********************************************************/
/* FILE:          LEVEL2.SRC
   VERSION:       KLINEF 5-20-82
   PROCEDURES
       DEFINED:     GATE$KEEPER          CREATE$EVC
                    READ                 AWAIT
                    ADVANCE              PREEMT
                    TICKET               CREATE$PROC
                    OUT$CHAR             OUT$LINE
                    OUT$NUM              OUT$DNUM
                    SEND$CHAR            OUT$HEX
                    RECV$CHAR            IN$CHAR
                    IN$NUM               IN$DNUM
                    IN$HEX

   REMARKS:  !!! CAUTION !!! !!! CAUTION !!! !!! CAUTION!!!
             IF NEW USER SERVICES ARE ADDED TO THIS MODULE
             OR CHANGES ARE MADE TO EXISTING ONES, MAKE
             SURE THE LOCATOR MAP (FILE: KORE.MP2) IS CHECK-
             ED TO SEE IF THE LOCATION OF 'GATE$KEEPER' HAS
             NOT CHANGED.  THE ABSOLUTE ADDRESS OF THIS
             PROCEDURE HAS BEEN SUPPLIED TO THE GATE$MODULE
             IN FILE: GATE.SRC.  IF IT HAS CHANGED THE NEW
             ADDRESS SHOULD BE UPDATED IN FILE:  GATE.SRC
             AND RECOMPILED.  ALL USER PROCESSES WILL HAVE
             TO BE RELINKED WITH FILE:  GATE.OBJ AND
             RELOCATED.

             LITERAL DECLARATIONS GIVEN AT THE BEGINNING
             OF SEVERAL MODULES ARE LOCAL TO THE ENTIRE
             MODULE.  HOWEVER, SOME ARE LISTED THE SAME
             IN MORE THAN ONE MODULE.  THE VALUE AND
             THEREFORE THE MEANING OF THE LITERAL IS
             COMMUNICATED ACROSS MODULE BOUNDARIES.
             'NOT$FOUND' USED IN LOCATE$EVC AND
             CREATE$EVC IS AN EXAMPLE.  TO CHANGE IT IN
             ONE MODULE AND NOT THE OTHER WOULD KILL
             THE CREATION OF ANY NEW EVENTCOUNTS BY THE
             OS.

                                                            */
/**************************************************************/
```

```
/*0073***************************************************/

L2$MODULE: DO;


/*****************************************************************/
/*****************************************************************/
/* LOCAL DECLARATIONS                                            */

DECLARE
   MAX$CPU                  LITERALLY      '10',
   MAX$VPS$CPU              LITERALLY      '10',
   MAX$CPU$$$$MAX$VPS$CPU   LITERALLY     '100',
   FALSE                    LITERALLY       '0',
   READY                    LITERALLY       '1',
   RUNNING                  LITERALLY       '3',
   WAITING                  LITERALLY       '7',
   TRUE                     LITERALLY     '119',
   NOT$FOUND                LITERALLY     '255',
   PORT$CC                  LITERALLY  '00CCH',
   RESET                    LITERALLY       '0',
   INT$RETURN               LITERALLY     '77H';

/*0096***************************************************/
/* PROCESSOR DATA SEGMENT TABLE                                  */
/*    DELARED PUBLIC IN MODULE 'L1$MODULE'                       */
/*                    IN FILE   'LEVEL1    '                     */

DECLARE PRDS STRUCTURE
  (CPU$NUMBER               BYTE,
   VP$START                 BYTE,
   VP$END                   BYTE,
   VPS$PER$CPU              BYTE,
   COUNTER                  WORD)          EXTERNAL;

/*0108***************************************************/
/* GLOBAL DATA BASE DECLARATIONS                                 */
/*    DECLARED PUBLIC IN FILE   'GLOBAL.SRC'                     */
/*                     IN MODULE'GLOBAL$MODULE'                  */
```

```
DECLARE VPM( MAX$CPU$$$$MAX$VPS$CPU ) STRUCTURE
  (VP$ID                    BYTE,
   STATE                    BYTE,
   VP$PRIORITY              BYTE,
   EVC$THREAD               BYTE,
   EVC$AW$VALUE             WORD,
   SS$REG                   WORD)          EXTERNAL;

DECLARE
   EVENTS                   BYTE           EXTERNAL;

DECLARE EVC$TBL (100) STRUCTURE
  (EVC$NAME                 BYTE,
   VALUE                    WORD,
   THREAD                   BYTE)          EXTERNAL;

DECLARE
   SEQUENCERS               BYTE           EXTERNAL;

DECLARE SEQ$TABLE (100) STRUCTURE
  (SEQ$NAME                 BYTE,
   SEQ$VALUE                WORD)          EXTERNAL;

DECLARE
   NR$VPS( MAX$CPU )        BYTE           EXTERNAL,
   NR$RPS                   BYTE           EXTERNAL,
   HDW$INT$FLAG (MAX$CPU )BYTE             EXTERNAL,
   GLOBAL$LOCK              BYTE           EXTERNAL;

/*0166***************************************************/
/* DECLARATION OF EXTERNAL PROCEDURE REFERENCES       */
/*    DECLARED PUBLIC IN FILE  'LEVEL1.SRC'           */
/*                    IN MODULE 'LEVEL1$MODULE'       */

VPSCHEDULER:  PROCEDURE EXTERNAL; END;
   /* IN FILE 'SCHED.ASM' */

RET$VP      :   PROCEDURE BYTE EXTERNAL; END;

LOCATE$EVC  :   PROCEDURE (EVENT$NAME) BYTE EXTERNAL;
   DECLARE EVENT$NAME BYTE;
END;

LOCATE$SEQ  :   PROCEDURE (SEQ$NAME) BYTE EXTERNAL;
   DECLARE SEQ$NAME BYTE;
END;
```

```
/*0205*************************************************************/
/* DIAGNOSTIC MESSAGES (WILL EVENTUALLY BE REMOVED)               */

DECLARE
   MSG16(*) BYTE INITIAL('ENTERING  PREEMT',13,10,'%'),
   MSG17(*) BYTE INITIAL('ISSUING   INTERRUPT!!',13,10,'%'),
   MSG18(*) BYTE INITIAL('ENTERING  AWAIT',10,13,'%'),
   MSG19(*) BYTE INITIAL('ENTERING  ADVANCE ',10,13,'%'),
   MSG21(*) BYTE INITIAL('ENTERING  CREATE$EVC FOR %'),
   MSG23(*) BYTE INITIAL('ENTERING  READ FOR EVC:  $'),
   MSG24(*) BYTE INITIAL('ENTERING  TICKET',13,10,'%'),
   MSG25(*) BYTE INITIAL('ENTERING  CREATE$SEQ  %'),
   MSG26(*) BYTE INITIAL('ENTERING  CREATE$PROC',10,13,'%'),
   MSG27(*) BYTE INITIAL(10,'ENTERING  GATE$KEEPER  N= %');

DECLARE
   CR LITERALLY  '0DH',
   LF LITERALLY  '0AH';

/*************************************************************/
```

```
/*0231*************************************************************/
/*****************************************************************/
/**  GATE$KEEPER   PROCEDURE              KLINEF 5-18-82 ****/
/*****************************************************************/
/*  THIS PROCEDURE IS THE ENTRY INTO THE OPERATING               */
/*  SYSTEM DOMAIN FROM THE USER DOMAIN.  THIS IS THE             */
/*  ACCESS POINT TO THE UTILITY/SERVICE ROUTINES AVAIL-          */
/*  ABLE TO THE USER. THIS PROCEDURE IS CALLED BY THE            */
/*  GATE MODULE WHICH IS LINKED WITH THE USER PROGRAM.           */
/*  IT IS THE GATE MODULE WHICH PROVIDES TRANSLATION             */
/*  FROM THE USER DESIRED FUNCTION TO THE FORMAT REQUIR-         */
/*  ED FOR THE GATEKEEPER.  THE GATEKEEPER CALLS THE             */
/*  DESIRED UTILITY/SERVICE PROCEDURE IN LEVEL2 OF THE           */
/*  OPERATING SYSTEM AGAIN PERFORMING THE NECESSARY              */
/*  TRANSLATION FOR A PROPER CALL.  THE TRANSLATIONS ARE         */
/*  INVISIBLE TO THE USER.  THE GATEKEEPER ADDRESS IS            */
/*  PROVIDED TO THE GATE MODULE TO BE USED FOR THE IN-           */
/*  DIRECT CALL.                                                 */
```

```
/*                                                              */
/*  THE PARAMETER LIST IS PROVIDED FOR CONVENIENCE AND          */
/*  REPRESENTS NO FIXED MEANING, EXCEPT FOR 'N'.                */
/*     N       FUNCTION CODE PROVIDED BY GATE                   */
/*     BYT     BYTE VARIABLE FOR TRANSLATION                    */
/*     WORDS   WORD                                             */
/*     PTR     POINTER VARIABLE FOR TRANSLATION                 */
/*0278*************************************************************/

GATE$KEEPER: PROCEDURE(N, BYT, WORDS, PTR) REENTRANT PUBLIC;

   DECLARE
     (N, BYT) BYTE,
      WORDS WORD,
      PTR POINTER;

/* I-O SERVICES ARE NOT ACKNOWLEDGED FOR TWO REASONS:           */
/*    1.  THEY ARE CALLED SO OFTEN THAT DIAGNOSTIC OUTPUT       */
/*        WOULD BE TOO CLUTTERED.                               */
/*    2.  THEY THEMSELVES PRODUCES I-O EFFECTS THAT             */
/*        ACKNOWLEDGE THEY ARE BEING CALLED.                    */
   IF N < 8 THEN DO;
      CALL OUT$LINE(@MSG27);
      CALL OUT$NUM(N);
      CALL OUT$CHAR(CR);
      CALL OUT$CHAR(LF);
   END;
   DO CASE N;                          /*  N  */
      CALL AWAIT(BYT,WORDS);           /*  0  */
      CALL ADVANCE(BYT);               /*  1  */
      CALL CREATE$EVC(BYT);            /*  2  */
      CALL CREATE$SEQ(BYT);            /*  3  */
      CALL TICKET(BYT,PTR);            /*  4  */
      CALL READ(BYT,PTR);              /*  5  */
      CALL CREATE$PROC(PTR);           /*  6  */
      CALL PREEMPT( BYT );             /*  7  */
      CALL OUT$CHAR(BYT);              /*  8  */
      CALL OUT$LINE(PTR);              /*  9  */
      CALL OUT$NUM(BYT);               /* 10  */
      CALL OUT$DNUM(WORDS);            /* 11  */
      CALL IN$CHAR(PTR);               /* 12  */
      CALL IN$NUM(PTR);                /* 13  */
      CALL IN$DNUM(PTR);               /* 14  */
   END;   /* CASE */
   RETURN;
END;   /* GATE$KEEPER */
```

```
/*0337*************************************************************/
/*  CREATE$EVC  PROCEDURE                          KLINEF 5-18-82 */
/*-------------------------------------------------------------*/
/* CREATES EVENTCOUNT FOR INTER-PROCESS SYNCHRONIZATION.  */
/* EVENTCOUNT IS INITIALIZED TO 0 IN THE EVENTCOUNT TABLE.*/
/*************************************************************/

CREATE$EVC: PROCEDURE(NAME) REENTRANT PUBLIC;

DECLARE NAME BYTE;

   CALL OUT$LINE(@MSG21);
   CALL OUT$NUM(NAME);
   CALL OUT$CHAR(CR);
   CALL OUT$CHAR(LF);

   /* ASSERT GLOBAL LOCK */
   DO WHILE LOCKSET(@GLOBAL$LOCK,119); END;

   IF /* THE EVENTCOUNT DOES NOT ALREADY EXIST */
      LOCATE$EVC(NAME) = NOT$FOUND THEN DO;
      /* CREATE THE EVENTCOUNT ENTRY BY ADDING THE  */
      /* NEW EVENTCOUNT TO THE END OF THE EVC$TABLE */
      EVC$TBL(EVENTS).EVC$NAME = NAME;
      EVC$TBL(EVENTS).VALUE = 0;
      EVC$TBL(EVENTS).THREAD = 255;
      /* INCREMENT THE SIZE OF THE EVC$TABLE */
      EVENTS = EVENTS + 1;
   END; /* CREATE THE EVENTCOUNT */
   /* RELEASE THE GLOBAL LOCK */
   GLOBAL$LOCK = 0;
   RETURN;
END; /* CREATE$EVC PROCEDURE */
```

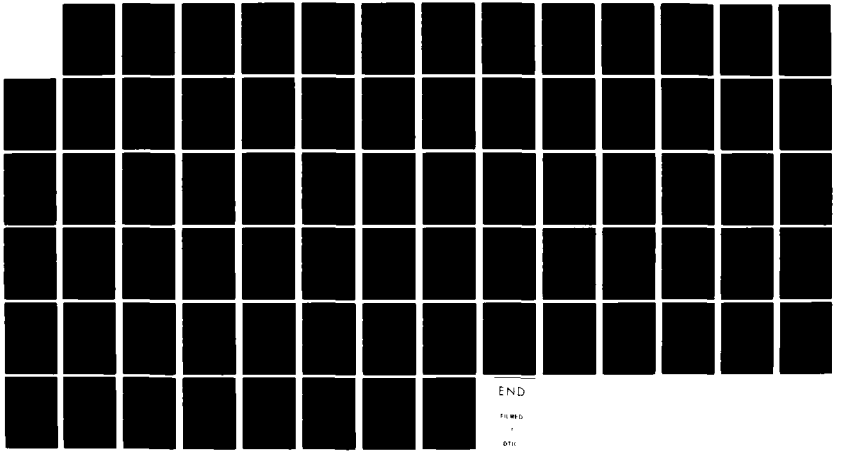
```
/*0403***************************************************/
/*   READ PROCEDURE                          KLINEF 5-19-82  */
/*---------------------------------------------------------*/
/* THIS PROCEDURE ALLOWS USERS TO READ THE PRESENT VALUE  */
/* OF THE SPECIFIED EVENT$COUNT WITHOUT MAKING ANY        */
/* CHANGES.  A POINTER IS PASSED TO PROVIDE A BASE TO A   */
/* VARIABLE IN THE CALLING ROUTINE FOR PASSING THE RETURN */
/* VALUE BACK TO THE CALLING ROUTINE.                     */
/***************************************************/

READ:  PROCEDURE( EVC$NAME, RETS$PTR ) REENTRANT PUBLIC;

DECLARE
   EVC$NAME              BYTE,
   EVCTBL$INDEX          BYTE,
   RETS$PTR              POINTER,
   EVC$VALUE$RET         BASED RETS$PTR WORD;

   /*  SET THE GLOBAL LOCK  */
   DO WHILE LOCKSET(@GLOBAL$LOCK,119);  END;

   CALL OUT$LINE(@MSG23);
   CALL OUT$NUM(EVC$NAME);
   CALL OUT$CHAR(CR);
   CALL OUT$CHAR(LF);

   /*  OBTAIN INDEX  */
   EVCTBL$INDEX = LOCATE$EVC( EVC$NAME );

   /*  OBTAIN VALUE  */
   EVC$VALUE$RET = EVC$TBL( EVCTBL$INDEX ).VALUE;

   /*  UNLOCK GLOBAL LOCK  */
   GLOBAL$LOCK = 0 ;
   RETURN;
END;   /*  READ PROCEDURE  */
```

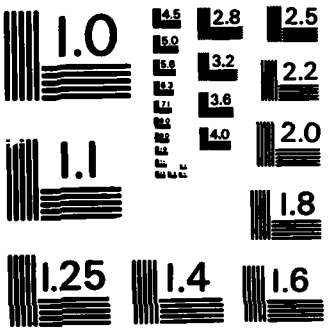
```
/*0469*************************************************************/
/*   AWAIT PROCEDURE                                               */
/*-------------------------------------------------------------*/
/* INTER PROCESS SYNCHRONIZATION PRIMITIVE.  SUSPENDS          */
/* EXECUTION OF RUNNING PROCESS UNTIL THE EVENTCOUNT HAS       */
/* REACHED THE SPECIFIED THRESHOLD VALUE, "AWAITED$VALUE."     */
/* USED BY THE OPERATING SYSTEM FOR THE MANAGEMENT OF          */
/* SYSTEM RESOURCES.                                           */
/*************************************************************/

AWAIT: PROCEDURE(EVC$ID,AWAITED$VALUE) REENTRANT PUBLIC;

DECLARE
   AWAITED$VALUE     WORD,
  (EVC$ID, NEED$SCHED, RUNNING$VP,EVCTBL$INDEX) BYTE;

   CALL OUT$LINE(@MSG18);

   /* LOCK GLOBAL LOCK */
   DO WHILE LOCK$SET(@GLOBAL$LOCK, 119);  END;
   NEED$SCHED = TRUE;

   /*  DETERMINE THE RUNNING VIRTUAL PROCESSOR  */
   RUNNING$VP = RET$VP;

   /*  GET EVC INDEX  */
   EVCTBL$INDEX = LOCATE$EVC(EVC$ID);
   /* DETERMINE IF CURRENT VALUE IS LESS THAN THE
      AWAITED VALUE */
   IF EVC$TBL(EVCTBL$INDEX).VALUE < AWAITED$VALUE THEN DO;
      /* BLOCK PROCESS */
      VPM(RUNNING$VP).STATE = WAITING;
      VPM(RUNNING$VP).EVC$THREAD=EVC$TBL(EVCTBL$INDEX).THREAD;
      VPM(RUNNING$VP).EVC$AW$VALUE = AWAITED$VALUE;
      EVC$TBL( EVCTBL$INDEX ).THREAD = RUNNING$VP;
      END;     /* BLOCK PROCESS */
   ELSE       /* DO NOT BLOCK PROCESS */
      NEED$SCHED = FALSE;

   /*  SCHEDULE THE VIRTUAL PROCESSOR  */
   IF NEED$SCHED = TRUE  THEN
      CALL VPSCHEDULER;              /* NO RETURN */

   /* UNLOCK GLOBAL LOCK */
   GLOBAL$LOCK = 0;
   RETURN;
END;   /* AWAIT PROCEDURE */
```

```
/*0535*************************************************************/
/*    ADVANCE   PROCEDURE                    KLINEF 5-19-82      */
/*-------------------------------------------------------------- -*/
/*  INTER PROCESS SYNCHRONIZATION PRIMITIVE.  INDICATES          */
/*  SPECIFIED EVENT HAS OCCURED BY ADVANCING(INCREMENTING)       */
/*  THE ASSOCIATED EVENTCOUNT.  EVENT IS BROADCAST TO ALL        */
/*  VIRTUAL PROCESSORS AWAITING THAT EVENT.                      */
/*-------------------------------------------------------------*/
/*  CALLS MADE TO:  OUT$LINE                                    */
/*                  VPSCHEDULER (NO RETURN)                     */
/***************************************************************/

ADVANCE: PROCEDURE(EVC$ID) REENTRANT PUBLIC;

DECLARE
  (EVC$ID, NEED$SCHED, NEED$INTR, EVCTBL$INDEX) BYTE,
  (SAVE, RUNNING$VP, I, CPU)                    BYTE;

   CALL OUT$LINE(@MSG19);

   /* LOCK THE GLOBAL LOCK */
   DO WHILE LOCKSET(@GLOBAL$LOCK,119);  END;

   RUNNING$VP = RET$VP;
   EVCTBL$INDEX = LOCATE$EVC(EVC$ID);
   EVC$TBL(EVCTBL$INDEX).VALUE=EVC$TBL(EVCTBL$INDEX).VALUE + 1;
   NEED$SCHED = FALSE;
   NEED$INTR = FALSE;
   SAVE = 255;
   I = EVC$TBL( EVCTBL$INDEX ).THREAD;
   DO WHILE  I <> 255;
      IF VPM(I).EVC$AW$VALUE <= EVC$TBL(EVCTBL$INDEX).VALUE
         THEN DO;       /* AWAKEN THE PROCESS */
         VPM(I).STATE = READY;
         VPM(I).EVC$AW$VALUE = 0;
         CPU = I / MAX$VPS$CPU ;
         IF SAVE = 225 THEN DO; /*THIS FIRST ONE IN LIST*
            EVC$TBL(EVCTBL$INDEX).THREAD=VPM(I).EVC$THREAD;
            VPM( I ).EVC$THREAD = 255;
            I = EVC$TBL( EVCTBL$INDEX ).THREAD;
            END;  /* IF FIRST */
         ELSE DO;       /* THEN THIS NOT FIRST IN LIST *
           VPM( SAVE ).EVC$THREAD = VPM( I ).EVC$THREAD
            VPM( I ).EVC$THREAD = 255;
            I = VPM( SAVE ).EVC$THREAD;
         END;  /* IF NOT FIRST */
         IF  ( CPU <> PRDS.CPU$NUMBER   THEN
            HDW$INT$FLAG( CPU ) = TRUE;
            NEED$INTR = TRUE;
            END;
```

AD-A121 878 IMPLEMENTATION OF A REAL-TIME DISTRIBUTED OPERATING SYSTEM FOR A MULTIPLE COMPUTER SYSTEM(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA S G KLINEFELTER JUN 82 2/2

UNCLASSIFIED F/G 9/2 NL





```
            ELSE NEED$SCHED = TRUE;
            END;      /* IF AWAKEN */
         ELSE DO;       /* DO NOT AWAKEN THIS PROCESS */
            SAVE = I;
            I = VPM( I ).EVC$THREAD;
         END;       /* IF NOT AWAKEN */
      END;       /* DO WHILE */
      IF NEED$INTR = TRUE    THEN DO;   /* HARDWARE INTR */
         CALL OUT$LINE( @MSG17 );
         DISABLE;
         OUTPUT(PORT$CC) =  80H;
         CALL TIME(1);
         OUTPUT(PORT$CC) = RESET;
         ENABLE;
      END;  /* NEED$INTR */
      IF NEED$SCHED = TRUE  THEN DO;
         VPM(RUNNING$VP).STATE = READY;
         CALL VPSCHEDULER;          /* NO RETURN */
      END;  /* IF NEED$SCHED */
      /* UNLOCK THE GLOBAL LOCK */
      GLOBAL$LOCK = 0;
      RETURN;
   END;  /* ADVANCE  PROCEDURE */
```

```
/*0667**************************************************************/
/*  PREEMT  PROCEDURE                              KLINEF 5-19-82  */
/*--------------------------------------------------------------*/
/* THIS PROCEDURE AWAKENS A HI PRIOITY PROCESS LEAVING          */
/* THE CURRENT RUNNING PROCESS IN THE  READY STATE AND          */
/* CALLS FOR A RESCHEDULING.  THE HIGH PRIORITY PROCESS         */
/* SHOULD BLOCK ITSELF WHEN FINISHED.                           */
/*   IF THE VP$ID IS 'FF' OR THE MONITOR PROCESS, IT WILL       */
/* MAKE IT READY WHERE-EVER IT IS IN THE VPM.  THE FOLLOW-      */
/* ING CODE DOES NOT TAKE ADVANTAGE OF THE FACT THAT            */
/* CURRENTLY IT IS THE THIRD ENTRY IN THE VPM FOR EACH          */
/* REAL PROCESOR.                                               */
/*--------------------------------------------------------------*/
/* CALLS MADE TO:  OUTLINE, VPSCHEDULER                         */
/**************************************************************/

PREEMPT:  PROCEDURE( VP$ID ) REENTRANT PUBLIC;

   DECLARE (VP$ID,SEARCH$ST,SEARCH$END,CPU,INDEX) BYTE;

   CALL OUT$LINE( @MSG16 );
   IF VP$ID <> 0FEH THEN DO; /* NORMAL PREEMT */
     /* SEARCH VPM FOR INDEX FOR ID  */
      SEARCH$ST = 0;
      DO CPU = 0 TO (NR$RPS - 1);
         SEARCH$END = SEARCH$ST + NR$VPS( CPU ) - 1 ;
         DO INDEX = SEARCH$ST TO SEARCH$END;
            IF VPM( INDEX ).VP$ID = VP$ID  THEN GO TO FOUND;
         END; /* DO INDEX */
         SEARCH$ST = SEARCH$ST + MAX$VPS$CPU;
      END; /* DO CPU */
      /* CASE IF NOT FOUND IS NOT ACCOUNTED FOR CURRENTLY */
      FOUND:
         /* LOCK THE GLOBAL LOCK */
         DO WHILE LOCK$SET(@GLOBAL$LOCK,119);  END;
         /* SET PREEMPTED VP TO READY */
         VPM( INDEX ).STATE = READY;
         /* NEED HARDWARE INTR OR RE-SCHED  */
         IF ( CPU = PRDS.CPU$NUMBER ) THEN DO;
            INDEX = RET$VP;    /* DETERMINE RUNNING PROCESS */
            VPM( INDEX ).STATE = READY;  /* SET TO READY */
            CALL VPSCHEDULER;    /* NO RETURN */
            END;
         ELSE DO;       /* CAUSE HARDWARE INTERRUPT */
            CALL OUT$LINE(@MSG17);
            HDW$INT$FLAG( CPU ) = TRUE;
            DISABLE;  OUTPUT( PORT$CC ) =  80H;
            CALL TIME(1);
            OUTPUT( PORT$CC ) = RESET;  ENABLE;
         END;
   END; /* NORMAL PREEMT */
```

```
   ELSE DO; /* PREEMT THE MONITOR */
      /* SEARCH VPM FOR ALL ID'S OF ØFEH */
      SEARCH$ST  = Ø;
      DO WHILE LOCK$SET(@GLOBAL$LOCK,119); END;
      DO CPU = Ø TO (NR$RPS - 1);
         SEARCH$END = SEARCH$ST + NR$VPS( CPU ) - 1;
         /* SET ALL INT$FLAGS EXCEPT THIS CPU'S */
         IF PRDS.CPU$NUMBER <> CPU THEN
            HDW$INT$FLAG( CPU ) = TRUE;
         DO INDEX = SEARCH$ST TO SEARCH$END;
            IF VPM( INDEX ).VP$ID = VP$ID THEN
               VPM( INDEX ).STATE = READY;
         END; /* DO */
         SEARCH$ST = SEARCH$ST + MAX$VPS$CPU;
      END; /* ALL MONITOR PROCESS SET TO READY */
      /* INTERRUPT THE OTHER CPU'S AND
      RESCHEDULE THIS ONE                    */
      CALL OUT$LINE(@MSG17);
      DISABLE;
      OUTPUT( PORT$CC ) = 80H;
      CALL TIME(1);
      OUTPUT( PORT$CC ) = RESET;
      ENABLE;
      INDEX = RET$VP;
      VPM(INDEX).STATE = READY;
      CALL VPSCHEDULER;  /* NO RETURN */
   END; /* ELSE
   /* UNLOCK GLOBAL MEMORY */
   GLOBAL$LOCK = Ø;
   RETURN;
END;   /*  PREEMPT PROCEDURE  */
```

```
/*0799*************************************************/
/*    CREATE$SEQ  PROCEDURE              KLINEF 5-20-82  */
/*-------------------------------------------------------*/
/* CREATOR OF INTER PROCESS SEQUENCER PRIMITIVES FOR USER */
/* PROGRAMS.  CREATES A SPECIFIED SEQUENCER AND INITIAL-  */
/* IZES IT TO 0, BY ADDING THE SEQUENCER TO THE END OF THE*/
/* SEQUENCER TABLE.                                       */
/*-------------------------------------------------------*/
/* CALLS MADE TO:  OUT$LINE          OUT$CHAR             */
/*                 OUT$HEX                                */
/*************************************************/

CREATE$SEQ: PROCEDURE(NAME) REENTRANT PUBLIC;

DECLARE NAME BYTE;

   /* ASSERT GLOBAL LOCK */
   DO WHILE LOCKSET(@GLOBAL$LOCK,119); END;

   CALL OUT$LINE(@MSG25);
   CALL OUT$HEX(NAME);
   CALL OUT$CHAR(CR);
   CALL OUT$CHAR(LF);

   IF /* THE SEQUENCER DOES NOT ALREADY EXIST, IE */
      LOCATE$SEQ(NAME) = NOT$FOUND THEN DO;
      /* CREATE THE SEQUENCER ENTRY BY ADDING THE  */
      /* NEW SEQUENCER TO THE END OF THE SEQ$TABLE */
      SEQ$TABLE(SEQUENCERS).SEQ$NAME   = NAME;
      SEQ$TABLE(SEQUENCERS).SEQ$VALUE  =  0;
      /*  INCREMENT NUMBER OF SEQUENCERS  */
      SEQUENCERS = SEQUENCERS + 1;
   END; /* CREATE THE SEQUENCER */
   /* RELEASE THE GLOBAL LOCK */
   GLOBAL$LOCK = 0;
   RETURN;
END; /* CREATE$SEQ PROCEDURE */
```

```
/*0865*************************************************************/
/*  TICKET   PROCEDURE                        KLINEF 5-20-82     */
/*-------------------------------------------------------------*/
/* INTER-VIRTUAL PROCESSOR SEQUENCER RPIMITIVE FOR  USER  */
/* PROGRAM.  SIMILAR TO "TAKE A NUMBER AND WAIT."  RETURNS*/
/* PRESENT VALUE OF SPECIFIED SEQUENCER AND INCREMENTS THE*/
/* SEQUENCER.  A POINTER IS PASSED TO PROVIDE A BASE TO A */
/* VARIABLE IN THE CALLING ROUTINE FOR PASSING THE RETURN */
/* VALUE BACK TO THE CALLING ROUTINE.                     */
/*-------------------------------------------------------------*/
/*  CALLS MADE TO:  OUT$LINE                                   */
/*************************************************************/

TICKET:  PROCEDURE( SEQ$NAME, RETS$PTR ) REENTRANT PUBLIC;

   DECLARE
      SEQ$NAME        BYTE,
      SEQTBL$INDEX    BYTE,
      RETS$PTR        POINTER,
      SEQ$VALUE$RET   BASED RETS$PTR WORD;

   /* ASSERT GLOBAL LOCK  */
   DO WHILE LOCKSET(@GLOBAL$LOCK,119);  END;

   CALL OUT$LINE(@MSG24);

   /*  OBTAIN SEQ$NAME INDEX  */
   SEQTBL$INDEX = LOCATE$SEQ( SEQ$NAME );
   /*  OBTAIN SEQUENCER VALUE  */
   SEQ$VALUE$RET = SEQ$TABLE( SEQTBL$INDEX ).SEQ$VALUE;
   /*  INCREMENT SEQUENCER  */
   SEQ$TABLE( SEQTBL$INDEX ).SEQ$VALUE  =
       SEQ$TABLE(SEQTBL$INDEX).SEQ$VALUE  +  1 ;

   /*  UNLOCK THE GLOBAL LOCK  */
   GLOBAL$LOCK  =  0 ;
   RETURN;
END;    /*  TICKET PROCEDURE  */
```

```
/*0931*************************************************************/
/*          CREATE$PROC    PROCEDURE     KLINEF  5-20-82          */
/*-------------------------------------------------------------*/
/*  THIS PROCEDURE CREATES A PROCESS FOR THE USER AS           */
/*  SPECIFIED BY THE INPUT PARAMETERS CONTAINED IN A           */
/*  STRUCTURE IN THE GATE MODULE.  THE PARAMETER PASSED        */
/*  IS A POINTER WHICH POINTS TO THIS STRUCTURE.               */
/*  INFO CONTAINED IN THIS STRUCTURE IS:  PROCESS ID,          */
/*  PROCESS PRIORITY, THE DESIRED PROC STACK LOCATION,         */
/*  AND THE PROCESS CODE STARTING LOCATION WHICH IS            */
/*  IS TWO ELEMENTS: THE IP REGISTER (OFFSET) AND THE          */
/*  CS REGISTER (CODE SEGMENT).                                */
/*-------------------------------------------------------------*/
/* CALLS MADE TO:   OUTLINE                                    */
/*************************************************************/

CREATE$PROC: PROCEDURE( PROC$PTR ) REENTRANT PUBLIC;

   DECLARE
      PROC$PTR        POINTER,
      PROC$TABLE BASED PROC$PTR STRUCTURE
        (PROC$ID              BYTE,
         PROC$PRI             BYTE,
         PROC$STACK$SEG       WORD,
         PROC$IP              WORD,
         PROC$CS              WORD);

   DECLARE
     (PS1, PS2)   WORD,
      TEMP        BYTE;

   DECLARE PROC$STACK$PTR POINTER AT(@PS1),
      PROC$STACK BASED PROC$STACK$PTR STRUCTURE
     (SP                   WORD,
      BP                   WORD,
      RET$TYPE             WORD,
      LENGTH(0FEH)         BYTE,
      DI                   WORD,
      SI                   WORD,
      DS                   WORD,
      DX                   WORD,
      CX                   WORD,
      AX                   WORD,
      BX                   WORD,
      ES                   WORD,
      IP                   WORD,
      CS                   WORD,
      FL                   WORD);

   CALL OUT$LINE(@MSG26);
```

```
    /*  TO SET UP PROC$STACK$PTR  */
    PS1 = 0;
    PS2 = PROC$TABLE.PROC$STACK$SEG;

    PROC$STACK.SP = 104H;
    PROC$STACK.BP = 0;
    PROC$STACK.RET$TYPE = INT$RETURN;
    PROC$STACK.DI = 0;
    PROC$STACK.SI = 0;
    PROC$STACK.DS = 0;
    PROC$STACK.DX = 0;
    PROC$STACK.CX = 0;
    PROC$STACK.AX = 0;
    PROC$STACK.BX = 0;
    PROC$STACK.ES = 0;
    PROC$STACK.IP = PROC$TABLE.PROC$IP;
    PROC$STACK.CS = PROC$TABLE.PROC$CS;
    PROC$STACK.FL = 200H;  /*SET IF FLAG (ENABLE INTR)*/

    /* SET GLOBAL LOCK */
    DO WHILE LOCKSET(@GLOBAL$LOCK,119);  END;

    IF PRDS.VPS$PER$CPU < MAX$VPS$CPU  THEN DO;
       TEMP = PRDS.VPS$PER$CPU + PRDS.VP$START;
       VPM( TEMP ).VP$ID = PROC$TABLE.PROC$ID;
       VPM( TEMP ).STATE = 01;       /* READY */
       VPM( TEMP ).VP$PRIORITY = PROC$TABLE.PROC$PRI;
       VPM( TEMP ).EVC$THREAD = 255;
       VPM( TEMP ).EVC$AW$VALUE = 0;
       VPM( TEMP ).SS$REG = PROC$TABLE.PROC$STACK$SEG;

       PRDS.VPS$PER$CPU = PRDS.VPS$PER$CPU + 1;
       PRDS.VP$END = PRDS.VP$END + 1;
       NR$VPS( PRDS.CPU$NUMBER ) =
          NR$VPS(PRDS.CPU$NUMBER) + 1;
    END;    /* DO */

    /* RELEASE THE GLOBAL LOCK */
    GLOBAL$LOCK = 0;
    RETURN;
END;        /*  CREATE$PROCESS  */
```

```
/*1063*************************************************************/
/*    IN$CHAR    PROCEDURE                    KLINEF 5-22-82      */
/*-------------------------------------------------------------*/
/* GETS A CHAR FROM THE SERIAL PORT.  CHAR IS !!!NOT!!!        */
/* ECHOED.  THAT IS RESPONSIBILTY OF USER IN THIS CASE.        */
/* INPUT TO SERIAL PORT VIA SBC861 DOWN LOAD PROGRAM MAY       */
/* NOT BE ACCEPTED.                                            */
/* POINTER IS PROVIDED BY USER SO HE CAN BE RETURNED THE       */
/* CHARACTER .                                                 */
/*-------------------------------------------------------------*/
/* CALLS MADE TO:  RECV$CAHR                                   */
/*************************************************************/

IN$CHAR:  PROCEDURE ( RET$PTR ) REENTRANT PUBLIC;

   DECLARE
      RET$PTR POINTER,
      INCHR BASED RET$PTR BYTE;

   DISABLE;
   INCHR = RECV$CHAR;
   ENABLE;
   RETURN;
END; /* IN$CHAR */

/*1094*************************************************************/
/*   IN$NUM      PROCEDURE              KLINEF  5-22-82          */
/*-------------------------------------------------------------*/
/*  GETS TWO ASCII CHAR FROM THE SERIAL PORT, EXAMINES         */
/*  THEM TO SEE IF THEY ARE IN THE SET 0..F HEX AND FORMS      */
/*  A BYTE VALUE.  EACH VALID HEX DIGIT IS ECHOED TO THE       */
/*  CRT.  IMPROPER CHAR ARE IGNORED.  NO ALLOWANCES ARE        */
/*  MADE FOR WRONG DIGITS.  GET IT RIGHT THE FIRST TIME.       */
/*  IF YOU ARE INDIRECTLY ACCESSING THE SERIAL PORT VIA        */
/*  THE SBC861 DOWN LOAD PROGRAM FROM THE MDS SYSTEM           */
/*  INPUT MAY NOT BE ACCEPTED.  A POINTER IS PASSED BY THE     */
/*  USER SO THAT HE RETURNED THE CHARACTER.                    */
/*-------------------------------------------------------------*/
/*  CALLS MADE TO:  IN$HEX                                     */
/*************************************************************/

IN$NUM:  PROCEDURE ( RET$PTR ) REENTRANT PUBLIC;
   DECLARE
      RET$PTR     POINTER,
      NUM BASED RET$PTR BYTE;

   DISABLE;
   NUM = IN$HEX;
   ENABLE;
   RETURN;
END; /* IN$NUM */
```

```
/*1129***************************************************************/
/*********************************************************************/
/*   OUT$CHAR      PROCEDURE                    KLINEF 5-20-82   */
/*-------------------------------------------------------------------*/
/* SENDS A BYTE TO THE SERIAL PORT                                   */
/*-------------------------------------------------------------------*/
/* CALL MADE TO:   SEND$CHAR                                         */
/*********************************************************************/

OUT$CHAR: PROCEDURE( CHAR ) REENTRANT PUBLIC;

   DECLARE CHAR BYTE;

   DISABLE;
   CALL SEND$CHAR( CHAR );
   ENABLE;
   RETURN;
END;
```

```
/*1153***************************************************************/
/*   OUT$LINE PROCEDURE                KLINEF  5-20-82          */
/*-------------------------------------------------------------------*/
/* USING A POINTER TO A BUFFER IT WILL OUTPUT AN ENTIRE              */
/* LINE THRU THE SERIAL PORT UNTIL AN '%' IS ENCOUNTERED             */
/* OR 80 CHARACTERS IS REACHED--WHICH EVER IS FIRST.  CR'S           */
/* AND LF'S CAN BE INCLUDED.                                         */
/*-------------------------------------------------------------------*/
/* CALLS MADE TO:  SEND$CHAR                                         */
/*********************************************************************/

OUT$LINE: PROCEDURE( LINE$PTR ) REENTRANT PUBLIC;

   DECLARE
      LINE$PTR POINTER,
      LINE BASED LINE$PTR (80) BYTE,
      II BYTE;

   DISABLE;
   DO II = 0 TO 79;
      IF LINE( II ) = '%' THEN  GO TO DONE;
      CALL SEND$CHAR( LINE( II ) );
   END;
   DONE:   ENABLE;
   RETURN;
END;
```

```
/*1195*************************************************************/
/*  OUT$NUM         PROCEDURE              KLINEF   5-20-82  */
/*-------------------------------------------------------------*/
/*  OUTPUTS A BYTE VAULE NUMBER THRU THE SERIAL PORT           */
/*-------------------------------------------------------------*/
/*  CALLS MADE TO:  OUT$HEX                                    */
/*************************************************************/

OUT$NUM: PROCEDURE( NUM ) REENTRANT PUBLIC;

   DECLARE NUM BYTE;

   DISABLE;
   CALL OUT$HEX( NUM );
   ENABLE;
   RETURN;
END;
```

```
/*1218*************************************************************/
/*  IN$DNUM         PROCEDURE              KLINEF   5-22-82  */
/*-------------------------------------------------------------*/
/* GETS FOUR ASCII FROM SERIAL PORT TO FORM WORD VALUE.        */
/* CRITERIA ARE THE SAME AS IN PROCEDURE IN$NUM.               */
/*-------------------------------------------------------------*/
/* CALLS MADE TO:  IN$HEX                                      */
/*************************************************************/

IN$DNUM:  PROCEDURE ( RET$PTR ) REENTRANT PUBLIC;

   DECLARE
      RET$PTR     POINTER,
      DNUM BASED RET$PTR WORD,
     (H, L)   WORD;

   DISABLE;
   H = IN$HEX;
   H = SHL( H, 8 );
   L  = IN$HEX;
   DNUM = (H OR L);
   ENABLE;
   RETURN;
END;
```

```
/*1261*************************************************************/
/*    OUT$DNUM     PROCEDURE            KLINEF 5-20-82            */
/*-------------------------------------------------------------*/
/*  OUTPUTS A WORD VALUE NUMBER VIA THE SERIAL PORT             */
/*-------------------------------------------------------------*/
/*  CALLS MADE TO:  OUT$HEX                                    */
/*************************************************************/

 OUT$DNUM: PROCEDURE( DNUM ) REENTRANT PUBLIC;

   DECLARE
      DNUM     WORD,
      SEND     BYTE;

   DISABLE;
   SEND = HIGH( DNUM );
   CALL OUT$HEX( SEND );
   SEND = LOW( DNUM );
   CALL OUT$HEX( SEND );
   ENABLE;
   RETURN;
END;
```

```
/*1289*************************************************************/
/*   RECV$CHAR   PROCEDURE                KLINEF 5-22-82          */
/*-------------------------------------------------------------*/
/* BOTTEM LEVEL PROCEDURE THAT OBTAINS A CHAR FROM THE           */
/* SERIAL PORT.  PARITY BIT IS REMOVED.  CHAR IS !!NOT!!         */
/* ECHOED.                                                       */
/*-------------------------------------------------------------*/
/* CALLS MADE TO:    NONE                                        */
/*************************************************************/

RECV$CHAR:  PROCEDURE BYTE REENTRANT PUBLIC;

   DECLARE
      CHR      BYTE;

   /*CHECH PORT STATUS BIT 2 FOR RECEIVE-READY SIGNAL */
   DO WHILE (INPUT(0DAH) AND 02H) = 0;  END;
   CHR = (INPUT(0D8H) AND 07FH);
   RETURN CHR;
END;
```

```
/*1327***************************************************************/
/*  SEND$CHAR     PROCEDURE           KLINEF     5-20-82  */
/*-------------------------------------------------------------------*/
/*  OUTPUTS A BYTE THRU THE SERIAL PORT.  THIS IS NOT A   */
/*  SERVICE AVAILABLE THRU THE GATEKEEPER BUT IT IS CALLED*/
/*  BY MANY OF THOSE PROCEDURES.  IT WILL STOP SENDING    */
/*  (AND EVERYTHING ELSE) IF IT SEES A ^S AT INPUT.  ^Q   */
/*  WILL RELEASE THE PROCEDURE TO CONTINUE.               */
/*  THE USER BEWARE!!!!! THIS IS ONLY A DIAGNOSTIC TOOL   */
/*  TO FREEZE THE CRT FOR STUDY.  RELEASING IT DOESN'T    */
/*  ASSURE NORMAL RESUMPTION OF EXECUTION.  (YOU MAY FORCE*/
/*  ALL BOARDS TO IDLE FOR EXAMPLE.)                      */
/*-------------------------------------------------------------------*/
/*  CALLS MADE TO:                                       */
/*******************************************************************/

SEND$CHAR: PROCEDURE(CHAR) REENTRANT PUBLIC;

   DECLARE (CHAR,INCHR) BYTE;

   /* CHECK PORT STATUS */
   INCHR = (INPUT(0D8H) AND 07FH);
   IF INCHR = 13H THEN
      DO WHILE (INCHR <> 11H);
         IF ((INPUT(0DAH) AND 02H) <> 0) THEN
            INCHR = (INPUT(0D8H) AND 07FH);
      END;
   DO WHILE (INPUT(0DAH) AND 01H) = 0;   END;
   OUTPUT(0D8H) = CHAR;
   RETURN;
END;
```

```
/*1393*************************************************************/
/*  IN$HEX    PROCEDURE                          KLINEF 5-22-82  */
/*-------------------------------------------------------------*/
/*  GETS 2 HEX CHAR FROM THE SERIAL PORT AND IGNORES ANY-      */
/*  THING ELSE.  EACH VALID HEX DIGIT IS ECHOED TO THE         */
/*  SERIAL PORT.  A BYTE VALUE IS FORMED FROM THE TWO HEX      */
/*  CHAR.                                                      */
/*-------------------------------------------------------------*/
/*  CALLS MADE TO:  RECV$CHAR                                  */
/*************************************************************/

IN$HEX:  PROCEDURE  BYTE REENTRANT PUBLIC;

   DECLARE
      ASCII(*) BYTE DATA ('0123456789ABCDEF'),
      ASCIIL(*) BYTE DATA('0123456789',61H,62H,63H,64H,65H,
         66H),
     (INCHR, HEXNUM, H, L)  BYTE,
      FOUND                 BYTE,
      STOP                  BYTE;

   /* GET HIGH PART OF BYTE */
   FOUND = 0;
   DO WHILE NOT FOUND;
      /* IF INVALID CHAR IS INPUT, COME BACK HERE */
      INCHR = RECV$CHAR;
      H = 0;
      STOP = 0;
```

```
      /* COMPARE CHAR TO HEX CHAR SET */
      DO WHILE NOT STOP;
         IF (INCHR=ASCII(H)) OR (INCHR = ASCIIL(H)) THEN DO;
            STOP = 0FFH;
            FOUND = 0FFH;
            CALL SEND$CHAR( INCHR ); /* TO ECHO IT */
            END;
         ELSE DO;
            H = H + 1;
            IF H = 10H THEN STOP = 0FFH;
         END; /* ELSE */
      END; /* DO WHILE */
      H = SHL( H, 4 );
   END; /* DO WHILE */
   FOUND = 0;
   /* GET LOW PART OF BYTE */
   DO WHILE NOT FOUND;
      /* AGAIN DO UNTIL VALID HEX CHAR IS INPUT */
      INCHR = RECV$CHAR;
      L = 0H;
      STOP = 0;
      DO WHILE NOT STOP;
         IF (INCHR=ASCII(L)) OR (INCHR=ASCIIL(L)) THEN DO;
            STOP = 0FFH;
            FOUND = 0FFH;
            CALL SEND$CHAR(INCHR);
            END;
         ELSE DO;
            L = L + 1;
            IF L = 10H THEN STOP = 0FFH;
         END; /* ELSE */
      END; /* DO WHILE */
   END; /* DO WHILE */
   RETURN (H OR L);
END; /* IN$HEX */
```

```
/*1525*************************************************************/
/*     OUT$HEX    PROCEDURE                          KLINEF 5-20-82 */
/*-------------------------------------------------------------------*/
/*  TRANSLATES BYTE VALUES TO ASCII CHARACTERS AND OUTPUTS*/
/*  THEM THRU THE SERIAL PORT                                        */
/*-------------------------------------------------------------------*/
/*  CALLS MADE TO:  SEND$CHAR                                        */
/*******************************************************************/

OUT$HEX: PROCEDURE(B) REENTRANT PUBLIC;

   DECLARE B BYTE;
   DECLARE ASCII(*) BYTE DATA ('0123456789ABCDEF');

   CALL SEND$CHAR(ASCII(SHR(B,4) AND 0FH));
   CALL SEND$CHAR(ASCII(B AND 0FH));
   RETURN;
END;

/*******************************************************************/

END;    /*  L2$MODULE  */

/*******************************************************************/
/*******************************************************************/
/*******************************************************************/
```

APPENDIX F

LEVEL I -- MCORTEX SOURCE CODE

All of the source code ir LEVEL I, except the scheduler and interrupt handler, is contained in file: LEVEL1.SRC. It is compiled with the LARGE attribute. The two exceptions are written in ASM86 and had to be listed in their own module. LEVEL I is one of the relocatable code modules in file: KORE.LNK. It is part of the executable code module in file: KORE. This module contains utility procedures used only by the operating system. The memory map for all of KORE is located at the end of this Appendix. The map comes from file: KORE.MP2.

```
/*****************************************************************/
/*****************************************************************/
/*****************************************************************/
/*  FILE:           LEVEL1.SRC
    VERSION:        KLINEF 5-25-82
    PROCEDURES
        DEFINED:    RET$VP          RDYTHISVP
                    GETWORK         LOCATE$EVC
                    LOCATE$SEQ      IDLE$PROC
                    MONITOR$PROC

    REMARKS:
       WARNING:  SEVERAL OF THE LITERAL DECLARATIONS BELOW
       HAVE A SIMILAR MEANING IN OTHER MODULES.  THAT MEAN-
       ING IS COMMUNICATED ACROSS MODULES BOUNDARIES.  BE
       CAREFUL WHEN CHANGING THEM.
                                                              */
/*****************************************************************/

L1$MODULE:  DO;

/*0024**************************************************************/
/*****************************************************************/
/*  LOCAL DECLARATIONS                                           */

DECLARE
   MAX$CPU                  LITERALLY     '10',
   MAX$VPS$CPU              LITERALLY     '10',
   MAX$CPU$$$MAX$VPS$CPU    LITERALLY    '100',
   FALSE                    LITERALLY      '0',
   READY                    LITERALLY      '1',
   RUNNING                  LITERALLY      '3',
   WAITING                  LITERALLY      '7',
   TRUE                     LITERALLY    '119',
   NOT$FOUND                LITERALLY    '255',
   PORT$C0                  LITERALLY  '00C0H',
   PORT$C2                  LITERALLY  '00C2H',
   PORT$CE                  LITERALLY  '00CEH',
   PORT$CC                  LITERALLY  '00CCH',
   RESET                    LITERALLY      '0',
   INT$RETURN               LITERALLY    '77H',
   IDLE$STACK$SEG           LITERALLY  '0310H',
   IDLE$STACK$ABS           LITERALLY '03100H',
   INIT$STACK$SEG           LITERALLY  '0320H',
   INIT$STACK$ABS           LITERALLY '03200H',
   MONITOR$STACK$SEG        LITERALLY  '0330H',
   MONITOR$STACK$ABS        LITERALLY '03300H';
```

```
/*0073*************************************************************/
/*  PROCESSOR DATA SEGMENT TABLE                                   */
/*    INFORMATION RELEVANT TO THE PARTICULAR PHYSICAL              */
/*    PROCESSOR ON WHICH IT IS RESIDENT.                           */
/*                                                                 */
/*    CPU$NUMBER:     UNIQUE SEQUENTIAL NUMBER ASSIGNED TO         */
/*                    THIS REAL PROCESSOR.                         */
/*    VP$START:       VPM INDEX OF THE FIRST VIRTUAL               */
/*                    PROCESS ASSIGNED TO THIS REAL CPU.           */
/*    VP$END:         INDEX IN VPM OF LAST VIRTUAL...              */
/*    VPS$PER$CPU:    THE NUMBER OF VP ASSIGNED TO THIS            */
/*                    REAL CPU.  MAX IS 10.                        */
/*    COUNTER:        AN ARBITRARY MEASURE OF PERFORMANCE.         */
/*                    COUNT MADE WHILE IN IDLE STATE.              */

DECLARE PRDS STRUCTURE
  (CPU$NUMBER       BYTE,
   VP$START         BYTE,
   VP$END           BYTE,
   VPS$PER$CPU      BYTE,
   COUNTER          WORD) PUBLIC INITIAL(0,0,0,0,0);

/*************************************************************/
/* GLOBAL DATA BASE DECLARATIONS                             */
/*    DECLARED PUBLIC IN FILE   'GLOBAL.SRC'                 */
/*                      IN MODULE 'GLOBAL$MODULE'            */

DECLARE VPM( MAX$CPU$$$$MAX$VPS$CPU ) STRUCTURE
  (VP$ID            BYTE,
   STATE            BYTE,
   VP$PRIORITY      BYTE,
   EVC$THREAD       BYTE,
   EVC$AW$VALUE     WORD,
   SS$REG           WORD) EXTERNAL;

DECLARE
   CPU$INIT         BYTE EXTERNAL,
   HDW$INT$FLAG( MAX$CPU ) BYTE EXTERNAL,
   NR$VPS( MAX$CPU ) BYTE EXTERNAL,
   NR$RPS           BYTE EXTERNAL,
   GLOBAL$LOCK      BYTE EXTERNAL;

DECLARE
   EVENTS BYTE EXTERNAL,
   EVC$TBL(100) STRUCTURE
      (EVC$NAME       BYTE,
       VALUE          WORD,
       THREAD         BYTE) EXTERNAL;
```

```
DECLARE
   SEQUENCERS BYTE EXTERNAL,
   SEQ$TABLE(100) STRUCTURE
       (SEQ$NAME     BYTE,
        SEQ$VALUE    WORD) EXTERNAL;

/*0145*************************************************************/
/* DECLARATION OF EXTERNAL PROCEDURE REFERENCES                    */
/*    THE FILE AND MODULE WHERE THEY ARE DEFINED ARE               */
/*    LISTED.                                                      */


INITIAL$PROC: PROCEDURE EXTERNAL;  END;
   /* IN FILE:    INITKK.SRC */
   /* IN MODULE:  INIT$MOD   */

AWAIT: PROCEDURE (EVC$ID,AWAITED$VALUE) EXTERNAL;
   DECLARE EVC$ID BYTE, AWAITED$VALUE WORD;
END;

VPSCHEDULER: PROCEDURE EXTERNAL;  END;
   /* IN FILE:    SCHED.ASM */

DECLARE INTVEC LABEL EXTERNAL;
   /* IN FILE:    SCHED.ASM */

DECLARE INTR$VECTOR POINTER AT(0110H) INITIAL(@INTVEC);
   /* IN FILE:    SCHED.ASM */

/*0168*************************************************************/
/*  THESE DIAGNOSTIC MESSAGES MAY EVENTUALLY BE REMOVED.  */
/*  THE UTILITY PROCEDURES, HOWEVER, ARE ALSO USED BY THE */
/*  MONITOR PROCESS.  THEY SHOULD NOT BE REMOVED.         */

DECLARE
   MSG1(*)  BYTE INITIAL ('ENTERING RET$VP  ',13,10,'%'),
   MSG1A(*) BYTE INITIAL ('   RUNNING$VP$INDEX =  %'),
   MSG4(*)  BYTE INITIAL ('ENTERING RDYTHISVP',13,10,'%'),
   MSG4A(*) BYTE INITIAL ('   SET VP TO READY:    VP =  %'),
   MSG7(*)  BYTE INITIAL ('ENTERING GETWORK',13,10,'%'),
   MSG7A(*) BYTE INITIAL ('   SET VP TO RUNNING:  VP =  %'),
   MSG7B(*) BYTE INITIAL ('   SELECTED$DBR = %'),
   MSG10(*) BYTE INITIAL ('ENTERING IDLE$VP ',13,10,'%'),
   MSG11(*) BYTE INITIAL ('UPDATE IDLE COUNT ',13,10,'%'),
   MSG12(*) BYTE INITIAL ('ENTERING KERNEL$INIT',10,13,'%'),
   MSG20(*) BYTE INITIAL ('ENTERING LOCATE$EVC ',10,13,'%'),
   MSG22(*) BYTE INITIAL ('ENTERING LOCATE$SEQ ',10,13,'%'),
   MSG23(*) BYTE INITIAL ('   FOUND',10,13,'%'),
   MSG24(*) BYTE INITIAL ('   NOT FOUND',10,13,'%');
```

```
DECLARE
   CR LITERALLY 'ØDH',
   LF LITERALLY 'ØAH';

OUT$CHAR: PROCEDURE( CHAR ) EXTERNAL;
   DECLARE CHAR BYTE;
END;

OUT$LINE: PROCEDURE( LINE$PTR ) EXTERNAL;
   DECLARE LINE$PTR POINTER;
END;

OUT$NUM: PROCEDURE( NUM ) EXTERNAL;
   DECLARE NUM BYTE;
END;

OUT$DNUM: PROCEDURE( DNUM ) EXTERNAL;
   DECLARE DNUM WORD;
   END;

OUT$HEX: PROCEDURE(B) EXTERNAL;
   DECLARE B BYTE;
END;

IN$CHAR: PROCEDURE ( RET$PTR )  EXTERNAL;
   DECLARE RET$PTR POINTER;
END;

IN$DNUM:  PROCEDURE (RET$PTR) EXTERNAL;
   DECLARE RET$PTR POINTER;
END;

IN$NUM: PROCEDURE (RET$PTR) EXTERNAL;
   DECLARE RET$PTR POINTER;
END;
```

```
/*0271*************************************************/
/* STACK DATA & INITIALIAZTION FOR SYSTEM PROCESSES      */


DECLARE IDLE$STACK  STRUCTURE
    (SP                 WORD,
     BP                 WORD,
     RET$TYPE           WORD,
     LENGTH(030H)       WORD,
     DI                 WORD,
     SI                 WORD,
     DS                 WORD,
     DX                 WORD,
     CX                 WORD,
     AX                 WORD,
     BX                 WORD,
     ES                 WORD,
     START             POINTER,  /* IP,CS */
     FL                 WORD) AT(IDLE$STACK$ABS)
          INITIAL( 66H, 0, INT$RETURN,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,        @IDLE$PROC,  200H );

DECLARE INIT$STACK  STRUCTURE
    (SP                 WORD,
     BP                 WORD,
     RET$TYPE           WORD,
     LENGTH(030H)       WORD,
     DI                 WORD,
     SI                 WORD,
     DS                 WORD,
     DX                 WORD,
     CX                 WORD,
     AX                 WORD,
     BX                 WORD,
     ES                 WORD,
     START             POINTER,   /* IP,CS */
     FL                 WORD) AT(INIT$STACK$ABS)
       INITIAL( 66H, 0, INT$RETURN,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
   0,0,0,0,0,0,0,0,     @INITIAL$PROC,   200H );
                    /* 200H SETS THE IF FLAG */
```

```
 DECLARE MONITOR$STACK STRUCTURE
    (SP                  WORD,
     BP                  WORD,
     RET$TYPE            WORD,
     LENGTH(030H)        WORD,
     DI                  WORD,
     SI                  WORD,
     DS                  WORD,
     DX                  WORD,
     CX                  WORD,
     AX                  WORD,
     BX                  WORD,
     ES                  WORD,
     START               POINTER,   /* IP,CS */
     FL                  WORD) AT(MONITOR$STACK$ABS)
        INITIAL(66H,0,INT$RETURN,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
      0,0,0,0,0,0,0,0,  @MONITOR$PROC, 200H);

/*0357*************************************************************/
/*************************************************************/
/*  RET$VP   PROCEDURE                          KLINEF 5-25-82  */
/*-----------------------------------------------------------*/
/* USED BY THE SCHEDULER TO FIND OUT WHAT IS THE CURRENT      */
/* RUNNING PROCESS.  IT'S INDEX IN VPM IS RETURNED.           */
/*-----------------------------------------------------------*/
/* CALLS MADE TO:  OUT$HEX      OUT$CHAR                      */
/*************************************************************/

RET$VP: PROCEDURE BYTE REENTRANT PUBLIC;

   DECLARE RUNNING$VP$INDEX BYTE;

   CALL OUT$LINE(@MSG1);

   /*  SEARCH THE VP MAP FOR RUNNING PROCESS INDEX */
   DO   RUNNING$VP$INDEX = PRDS.VP$START TO PRDS.VP$END;
      IF VPM( RUNNING$VP$INDEX ).STATE = RUNNING
      THEN GO TO FOUND;
   END; /* DO */

FOUND:
   CALL OUT$LINE(@MSG1A);
   CALL OUT$HEX(RUNNING$VP$INDEX);
   CALL OUT$CHAR(CR);
   CALL OUT$CHAR(LF);
   RETURN RUNNING$VP$INDEX;
END; /* RET$VP PROCEDURE */
```

```
/*0403***************************************************/
/*  RDYTHISVP     PROCEDURE              KLINEF 5-25-82   */
/*-------------------------------------------------------*/
/*  CHANGES A VIRTUAL PROCESSOR STATE TO READY           */
/*-------------------------------------------------------*/
/*  CALLS MADE TO:  OUT$HEX    OUT$CHAR                  */
/***************************************************/

RDYTHISVP: PROCEDURE REENTRANT PUBLIC;

   DECLARE VP  BYTE;

   CALL OUT$LINE(@MSG4);

   VP = RET$VP;

   CALL OUT$LINE(@MSG4A);
   CALL OUT$HEX(VP);
   CALL OUT$CHAR(CR);
   CALL OUT$CHAR(LF);
   VPM(VP).STATE = READY;
   RETURN;
END; /* RDYTHISVP PROCEDURE */
```

```
/*0469***************************************************/
/*  GETWORK     PROCEDURE                  KLINEF  5-25-82  */
/*-------------------------------------------------------*/
/*  DETERMINES THE NEXT ELIGIBLE VIRTUAL PROCESSOR TO RUN */
/*-------------------------------------------------------*/
/*  CALLS MADE TO:  OUT$CHAR   OUT$LINE   OUT$DNUM       */
/***************************************************/

GETWORK: PROCEDURE WORD REENTRANT PUBLIC;

   DECLARE (PRI,N,I)     BYTE;
   DECLARE SELECTED$DBR WORD;
   DECLARE DISPLAY       BYTE;

   CALL OUT$LINE(@MSG7);

   PRI = 255;

   DO /* SEARCH VPM FOR ELIGIBLE VIRTUAL PROCESSOR TO RUN */
      I = PRDS.VP$START TO PRDS.VP$END;
      IF /* THIS VP'S PRIORITY IS HIGHER THAN PRI */
         ((VPM(I).VP$PRIORITY <= PRI) AND
         (VPM(I).STATE = READY))  THEN  DO;
            /* SELECT THIS VIRTUAL PROCESSOR */
            PRI = VPM(I).VP$PRIORITY;
            N = I;
      END; /* IF */
   END; /* DO LOOP SEARCH OF VPM */

   /*  SET SELECTED VIRTUAL PROCESSOR  */
   VPM(N).STATE = RUNNING;
   SELECTED$DBR = VPM(N).SS$REG;

   CALL OUT$LINE(@MSG7A);
   CALL OUT$HEX(N);
   CALL OUT$CHAR(CR);
   CALL OUT$CHAR(LF);

   CALL OUT$LINE(@MSG7B);
   CALL OUT$DNUM(SELECTED$DBR);
   CALL OUT$CHAR(CR);
   CALL OUT$CHAR(LF);

   RETURN SELECTED$DBR;

END; /* GETWORK PROCEDURE */
```

```
/*0535*************************************************************/
/*  LOCATE$EVC PROCEDURE                    KLINEF 5-25-82   */
/*-------------------------------------------------------------*/
/* FUNCTION CALL.  RETURNS THE INDEX IN EVENTCOUNT TABLE  */
/* OF THE EVENT NAME PASSED TO IT.                            */
/*-------------------------------------------------------------*/
/* CALLS MADE TO:  OUT$CHAR   OUT$LINE                        */
/*************************************************************/

LOCATE$EVC: PROCEDURE(EVENT$NAME) BYTE REENTRANT PUBLIC;

   DECLARE EVENT$NAME BYTE;
   DECLARE (MATCH,EVCTBL$INDEX) BYTE;

   CALL OUT$LINE(@MSG20);

   MATCH = FALSE;
   EVCTBL$INDEX = 0;
   /* SEARCH DOWN THE EVENTCOUNT TABLE TO LOCATE THE  */
   /* DESIRED EVENTCOUNT BY MATCHING THE NAMES */
   DO WHILE (MATCH = FALSE)  AND  (EVCTBL$INDEX < EVENTS);
      /* DO WHILE HAVE NOT FOUND THE EVENTCOUNT AND HAVE NOT */
      /* REACHED END OF THE TABLE */
      IF EVENT$NAME = EVC$TBL(EVCTBL$INDEX).EVC$NAME THEN
         MATCH = TRUE;
      ELSE
         EVCTBL$INDEX = EVCTBL$INDEX+1;
   END;  /*  WHILE  */
   /* IF HAVE FOUND THE EVENTCOUNT */
   IF  (MATCH = TRUE)    THEN  DO;
      /* RETURN ITS INDEX IN THE EVC$TBL */
      CALL OUT$LINE(@MSG23);
      RETURN EVCTBL$INDEX;
      END;
   ELSE DO;
      /* RETURN NOT FOUND CODE */
      CALL OUT$LINE(@MSG24);
      RETURN NOT$FOUND;
   END; /* ELSE */
END;  /* LOCATE$EVC  PROCEDURE */
```

```
/*0601***************************************************************/
/*  LOCATE$SEQ     PROCEDURE          KLINEF   5-23-82      */
/*-------------------- ------------------------------------------*/
/* FUNCTION CALL TO RETURN THE INDEX OF THE SEQUENCER      */
/* SPECIFIED IN THE SEQ-TABLE.                             */
/*-------------------- ------------------------------------------*/
/* CALLS MADE TO:  OUT$LINE                                */
/***************************************************************/

LOCATE$SEQ:   PROCEDURE(SEQ$NAME) BYTE REENTRANT PUBLIC;

   DECLARE SEQ$NAME BYTE;
   DECLARE ( MATCH, SEQTBL$INDEX ) BYTE;

   CALL OUT$LINE(@MSG22);

   MATCH  =  FALSE;
   SEQTBL$INDEX = 0;
   DO WHILE (MATCH = FALSE) AND (SEQTBL$INDEX < SEQUENCERS);
      IF SEQ$NAME = SEQ$TABLE(SEQTBL$INDEX).SEQ$NAME THEN
         MATCH = TRUE;
      ELSE
         SEQTBL$INDEX = SEQTBL$INDEX + 1;
   END;    /*  WHILE  */
   IF (MATCH = TRUE)  THEN DO;
      CALL OUT$LINE(@MSG23);
      RETURN SEQTBL$INDEX;
      END; /* IF */
   ELSE DO;
      CALL OUT$LINE(@MSG24);
      RETURN NOT$FOUND;
   END; /* ESLE */
END;       /* LOCATE$SEQ PROCEDURE */
```

```
/*0667***************************************************************/
/*****************************************************************/
/*  SYSTEM PROCESSES                                             */
/*                                                               */
```

```
/*****************************************************************/
/*  IDLE PROCESS                              KLINEF  5-24-82    */
/*---------------------------------------------------------------*/
/*  THIS PROCESS IS SCHEDULED IF ALL OTHER PROCESSES IN          */
/*  THE VPM ARE BLOCKED.  THE STARTING ADDRESS IS PROVIDED       */
/*  TO THE IDLE$STACK AND PLACED IN PRDS.IDLE$DBR.  A            */
/*  COUNTER IS INCREMENTED ABOUT EVERY SECOND.  THE COUNT        */
/*  IS MAINTAINED IN THE PRDS TABLE AND IS A ROUGH MEASURE       */
/*  OF SYSTEM PERFORMANCE BY GIVING AN INDICATION OF THE         */
/*  AMOUNT OF TIME SPENT IN THE IDLE PROCESS.                    */
/*---------------------------------------------------------------*/
/*  CALLS MADE TO:  PLM86 PROCEDURE 'TIME'                       */
/*                  OUT$LINE                                     */
/*****************************************************************/

IDLE$PROC:  PROCEDURE REENTRANT PUBLIC;

   DECLARE I BYTE;

   CALL OUT$LINE(@MSG10);

   /*  DELAYS ONE (1) SECOND  */
LOOP: DO I = 1 TO 40;
         CALL TIME( 250 );
      END;

      CALL OUT$LINE(@MSG11);

      PRDS.COUNTER = PRDS.COUNTER + 1;
      GO TO LOOP;
END;   /* IDLE$PROC  */
```

```
/*0689*************************************************************/
/*  MONITOR PROCESS                                  KLINEF 5-26-82  */
/*--------------------------------------------------------------*/
/*  THE MONITOR PROCESS IS INITIALIZED BY THE OS LIKE           */
/*  INIT AND IDLE.  IT HAS THE RESERVED ID OF 0FEH AND A        */
/*  PRIORITY OF 0H.  IT IS ALWAYS BLOCKED OR WAITING UNTIL      */
/*  IT IS PREEMTED BY THE USER.                                 */
/*--------------------------------------------------------------*/
/*  CALLS MADE TO:  OUT$LINE        OUT$CHAR                    */
/*                  OUT$DNUM        IN$DNUM                     */
/*                  IN$NUM                                      */
/*************************************************************/

MONITOR$PROC:  PROCEDURE REENTRANT PUBLIC;

   DECLARE
      PTR                POINTER,
      PTR2               POINTER,
      PTR3 BASED PTR2    POINTER,
      ADDR STRUCTURE (OFFSET WORD, BASE WORD),
      CONTENTS BASED PTR BYTE;

   DECLARE
     (LINECOMPLETE, LOOP2)     BYTE,
     (QUANTITY, COUNT)         BYTE,
     (INCHR, INDEX, VALID$CMD) BYTE;

   LOOP:  VALID$CMD = 0;

      CALL OUT$CHAR(CR);
      CALL OUT$CHAR(LF);
      CALL OUT$CHAR('.');
      DO WHILE NOT VALID$CMD;
         CALL IN$CHAR(@INCHR);
         IF (INCHR = 'D') OR (INCHR = 'S') OR (INCHR = 'E') THEN
            VALID$CMD = 0FFH;
         IF (INCHR=64H) OR (INCHR=65H) OR (INCHR=73H) THEN
            VALID$CMD = 0FFH;
         IF VALID$CMD = 0FFH THEN CALL OUT$CHAR(INCHR);
      END; /* DO WHILE */
```

```
IF (INCHR = 'D') OR (INCHR = 64H) THEN DO;
   /* DISPLAY COMMAND SECTION */
   CALL IN$DNUM(@ADDR.BASE);
   CALL OUT$CHAR(':');
   CALL IN$DNUM(@ADDR.OFFSET);
   PTR2 = @ADDR;
   PTR = PTR3;
   /* CONTENTS SHOULD NOW BE SET */
   DO WHILE (INCHR<>CR) AND (INCHR<>23H);
      CALL IN$CHAR(@INCHR);
   END; /* DO WHILE */
   IF INCHR = CR THEN DO;
      CALL OUT$CHAR('-');
      CALL OUT$NUM(CONTENTS);
      CALL OUT$CHAR(CR);
      CALL OUT$CHAR(LF);
   END; /* IF NORMAL 1 ADDR DISPLAY */
   IF INCHR = 23H THEN DO;
      COUNT = 0;
      CALL OUT$CHAR('#');
      CALL IN$NUM(@QUANTITY);
      DO WHILE QUANTITY > 0;
         CALL OUT$CHAR(CR);
         CALL OUT$CHAR(LF);
         CALL OUT$DNUM(ADDR.BASE);
         CALL OUT$CHAR(':');
         CALL OUT$DNUM(ADDR.OFFSET);
         LINECOMPLETE = FALSE;
         DO WHILE LINECOMPLETE = FALSE;
            CALL OUT$CHAR(' ');
            CALL OUT$NUM(CONTENTS);
            ADDR.OFFSET = ADDR.OFFSET + 1;
            PTR = PTR3;
            QUANTITY = QUANTITY - 1;
            IF ((ADDR.OFFSET AND 000FH)=0) OR
               (QUANTITY = 0) THEN LINECOMPLETE=TRUE;
         END; /* DO WHILE LINE NOT COMPLETE */
      END; /* DO WHILE QUANTITY */
   END; /* IF MULTI ADDR DISPLAY */
END; /* DISPLAY COMMAND SECTION */
```

```
IF (INCHR='S') OR (INCHR=73H) THEN DO;
   /* SUBSTITUTE COMMAND SECTION */
   CALL IN$DNUM(@ADDR.BASE);
   CALL OUT$CHAR(':');
   CALL IN$DNUM(@ADDR.OFFSET);
   CALL OUT$CHAR('-');

   PTR2 = @ADDR;
   PTR = PTR3;
   /* CURRENT CONTENTS SHOULD NOW BE AVAILABLE */
   CALL OUT$NUM(CONTENTS);
   LOOP2 = TRUE;
   DO WHILE LOOP2 = TRUE;
      DO WHILE (INCHR<>',')AND(INCHR<>' ')
         AND(INCHR<>CR);
         CALL IN$CHAR(@INCHR);
      END;
      IF (INCHR = CR) THEN LOOP2 = FALSE;
      IF (INCHR = ',') THEN DO;
         /* SKIP THIS ADDR AND GO TO NEXT FOR SUB */
         CALL OUT$CHAR(CR);
         CALL OUT$CHAR(LF);
         ADDR.OFFSET = ADDR.OFFSET + 1;
         PTR = PTR3;
         CALL OUT$DNUM(ADDR.BASE);
         CALL OUT$CHAR(':');
         CALL OUT$DNUM(ADDR.OFFSET);
         CALL OUT$CHAR('-');
         CALL OUT$NUM(CONTENTS);
      END;  /* IF SKIP FOR NEXT SUB */
      IF (INCHR = ' ') THEN DO;
         CALL OUT$CHAR(' ');
         CALL IN$NUM(@CONTENTS);
         DO WHILE (INCHR<>CR)AND(INCHR<>',');
            CALL IN$CHAR(@INCHR);
         END;
         IF (INCHR = CR) THEN LOOP2 = FALSE;
         IF (INCHR = ',') THEN DO;
            CALL OUT$CHAR(',');
            ADDR.OFFSET = ADDR.OFFSET + 1;
            PTR = PTR3;
            CALL OUT$CHAR(CR);
            CALL OUT$CHAR(LF);
            CALL OUT$DNUM(ADDR.BASE);
            CALL OUT$CHAR(':');
            CALL OUT$DNUM(ADDR.OFFSET);
            CALL OUT$CHAR('-');
            CALL OUT$NUM(CONTENTS);
         END; /* IF GO TO NEXT ADDR */
      END; /* IF CHANGE CONTENTS */
      INCHR = 'X'; /* REINITIALIZE CMD */
```

```
            END; /* LOOP, CONTINUOUS SUB CMD */
         END; /* SUBSTITUTE COMMAND SECTION */

         IF (INCHR='E') OR (INCHR=65H) THEN DO;
            /* FIND OUT WHICH VPS IS RUNNING 'ME' */
            INDEX = RET$VP;
            /* NOW BLOCK MYSELF */
            VPM(INDEX).STATE = WAITING;
            CALL VPSCHEDULER; /* NO RETURN */
         END; /* IF */
      GO TC LOOP;
END; /* MONITOR PROCESS */
```

```
/*0997*************************************************************/
/*************************************************************/
/*          STARTING POINT OF THE OPERATING SYSTEM           */
/*-----------------------------------------------------------*/
/* ROUTINE INITIALIZES THE OS AND IS NOT REPEATED.           */
/*************************************************************/
/*************************************************************/

/* TO INITIALIZE THE PRDS TABLE FOR THIS CPU */
DECLARE CPU$PTR POINTER DATA(@PRDS.CPU$NUMBER),
            ZZ BYTE;

DISABLE;

CALL OUT$LINE(@MSG12);

/* INITIALIZE   P P I   AND   P I C */
OUTPUT(PORT$CE) = 80H;/* PPI CONTROL - MAKE PORT C OUTPUT */
OUTPUT(PORT$C0) = 13H;  /* PIC - ICW1 - EDGE TRIGGERED */
OUTPUT(PORT$C2) = 40H;/* PIC - ICW2 -VECTOR TABLE ADDRESS */
OUTPUT(PORT$C2) = 0FH;/* PIC - ICW4 -MCS86 MODE, AUTO EOI */


/* ESTABLISH UNIQUE SEQUENTIAL NUMBER FOR THIS CPU */
/* SET GLOBAL$LOCK */
DO WHILE LOCK$SET(@GLOBAL$LOCK,119);  END;
PRDS.CPU$NUMBER = CPU$INIT;
CPU$INIT = CPU$INIT + 1;

/* RELEASE GLOBAL LOCK */
GLOBAL$LOCK = 0;

/*  SET UP INITIAL START AND END FOR PROC TABLE */
PRDS.VP$START = 0;
DO ZZ = 1 TO PRDS.CPU$NUMBER;
   PRDS.VP$START = PRDS.VP$START + MAX$VPS$CPU;
END;
PRDS.VP$END = PRDS.VP$START + 2;
PRDS.VPS$PER$CPU = 3;

/* INITIALIZE THE VP MAP FOR IDLE AND INIT PROC */
/* AND MONITOR PROCESS */
VPM(PRDS.VP$START).VP$ID = 255;
VPM(PRDS.VP$START).STATE = 1;
VPM(PRDS.VP$START).VP$PRIORITY = 0;
VPM(PRDS.VP$START).EVC$THREAD = 255;
VPM(PRDS.VP$START).EVC$AW$VALUE = 0;
VPM(PRDS.VP$START).SS$REG = INIT$STACK$SEG;
```

```
VPM(PRDS.VP$START+1).VP$ID = 255;
VPM(PRDS.VP$START+1).STATE = 1;
VPM(PRDS.VP$START+1).VP$PRIORITY = 255;
VPM(PRDS.VP$START+1).EVC$THREAD = 255;
VPM(PRDS.VP$START+1).EVC$AW$VALUE = 0;
VPM(PRDS.VP$START+1).SS$REG = IDLE$STACK$SEG;

VPM(PRDS.VP$START+2).VP$ID = 0FEH;
VPM(PRDS.VP$START+2).STATE = 7;
VPM(PRDS.VP$START+2).VP$PRIORITY = 0;
VPM(PRDS.VP$START+2).EVC$THREAD = 255;
VPM(PRDS.VP$START+2).EVC$AW$VALUE = 0;
VPM(PRDS.VP$START+2).SS$REG = MONITOR$STACK$SEG;

NR$RPS = NR$RPS + 1;
NR$VPS(PRDS.CPU$NUMBER) = 3;

HDW$INT$FLAG( PRDS.CPU$NUMBER ) = 0 ;
ENABLE;

CALL VPSCHEDULER;          /* - - NO RETURN */

/*****************************************************************/
/*****************************************************************/

END;  /* L1$MODULE */

/*****************************************************************/
/*****************************************************************/
/*****************************************************************/
```

```
ISIS-II MCS-86 LOCATER, V1.1 INVOKED BY:
LOC86 KORE.LNK ADDRESSES(SEGMENTS(&
STACK(03000H),&
INITMOD_CODE(02800H),&
GLOBALMODULE_DATA(0F0000H)))&
SEGSIZE(STACK(75H))&
RS(0H TO 0FFFH) MAP
WARNING 56:  SEGMENT IN RESERVED SPACE
  SEGMENT:  (NO NAME)

SYMBOL TABLE OF MODULE L1MODULE
READ FROM FILE KORE.LNK
WRITTEN TO FILE :F0:KORE

BASE   OFFSET TYPE SYMBOL      BASE   OFFSET TYPE SYMBOL

0240H  000AH  PUB  PRDS        0100H  046AH  PUB  MONITORPROC
0100H  0420H  PUB  IDLEPROC    0100H  0381H  PUB  LOCATESEQ
0100H  02E2H  PUB  LOCATEEVC   0100H  0217H  PUB  GETWORK
0100H  01C0H  PUB  RDYTHISVP   0100H  014EH  PUB  RETVP
0182H  0BB5H  PUB  OUTHEX      0182H  0A3CH  PUB  INHEX
0182H  0A60H  PUB  SENDCHAR    0182H  0A3DH  PUB  RECVCHAR
0182H  0A11H  PUB  OUTDNUM     0182H  09D8H  PUB  INDNUM
0182H  09C0H  PUB  OUTNUM      0182H  0971H  PUB  OUTLINE
0182H  0959H  PUB  OUTCHAR     0182H  093EH  PUB  INNUM
0182H  0923H  PUB  INCHAR      0182H  07DEH  PUB  CREATEPROC
0182H  0772H  PUB  TICKET      0182H  06ECH  PUB  CREATESEQ
0182H  04E7H  PUB  PREEMPT     0182H  0334H  PUB  ADVANCE
0182H  027FH  PUB  AWAIT       0182H  020AH  PUB  READ
0182H  0180H  PUB  CREATEEVC   0182H  0060H  PUB  GATEKEEPER
0264H  0000H  PUB  VPSCHEDULER0264H  003DH  PUB  INTVEC
0280H  0002H  PUB  INITIALPROCE000H  0190H  PUB  VPM
E000H  04C9H  PUB  SEQTABLE    E000H  04C8H  PUB  SEQUENCERS
E000H  04C7H  PUB  CPUINIT     E000H  0000H  PUB  EVCTBL
E000H  04C6H  PUB  EVENTS      E000H  04BCH  PUB  HDWINTFLAG
E000H  04B2H  PUB  NRVPS       E000H  04B1H  PUB  NRRPS
E000H  04B0H  PUB  GLOBALLOCK

MEMORY MAP OF MODULE L1MODULE
READ FROM FILE KORE.LNK
WRITTEN TO FILE :F0:KORE
```

```
MODULE START ADDRESS  PARAGRAPH = 0100H  OFFSET = 0030H
SEGMENT MAP

START     STOP      LENGTH ALIGN NAME                CLASS

00110H    00113H     0004H   A   (ABSOLUTE)
01000H    01825H     0826H   W   L1MODULE_CODE      CODE
01826H    02408H     0BE3H   W   L2MODULE_CODE      CODE
0240AH    0240AH     0000H   W   GLOBALMODULE_C     CODE
                                 -ODE
0240AH    0253FH     0132H   W   L1MODULE_DATA      DATA
0253CH    02617H     00E4H   W   L2MODULE_DATA      DATA
02620H    0263DH     001EH   W   INITMOD_DATA       DATA
02640H    02640H     0000H   G   ??SEG
02640H    026E0H     00A1H   G   SCHEDULER
02800H    02824H     0025H   W   INITMOD_CODE       CODE
03000H    03074H     0075H   W   STACK              STACK
03100H    03173H     007CH   A   (ABSOLUTE)
03200H    0327FH     007CH   A   (ABSOLUTE)
03300H    0337BH     007CH   A   (ABSOLUTE)
E0000H    E05F4H     05F5H   W   GLOBALMODULE_D     DATA
                                 -ATA
E05F6H    E05F6H     0000H   W   MEMORY             MEMORY
```

APPENDIX G

SCHEDULER & INTERRUPT HANDLER SOURCE CODE

The source code in this appendix is part of LEVEL I, but it was assembled under ASM86 instead of of compiled. No special attributes are required for the assembler. The source code is contained in file: SCHED.ASM. It is linked along with the other modules of LEVEL I and LEVEL II into file: KORE.LNK. Its memory map is included in the memory map for KORE.

```
;*****************************************************************
;*   SCHEDULER     ASM FILE              KLINEF 2-27-82          *
;*---------------------------------------------------------------*
;* THE FOLLOWING ARE THE EXTERNAL PLM86 PROCEDURES CALLED        *
;* BY THIS MODULE.                                               *

   EXTRN GETWORK:FAR
   EXTRN RDYTHISVP:FAR
   EXTRN PRDS:BYTE
   EXTRN HDWINTFLAG:BYTE
   EXTRN GLOBALLOCK:BYTE

SCHEDULER SEGMENT

   PUBLIC VPSCHEDULER
   PUBLIC INTVEC

VPSCHEDULER PROC FAR

   ASSUME CS:SCHEDULER
   ASSUME DS:NOTHING
   ASSUME SS:NOTHING
   ASSUME ES:NOTHING

   ; ENTRY POINT FOR A CALL TO SCHEDULER

   CLI
   PUSH DS
   MOV CX,0H

   ; SWAP VIRTUAL PROCESSORS.  THIS IS ACCOMPLISHED BY
   ; SAVING THE SP AND BP REGISTERS ON THE STACK, ALONG
   ; WITH THE RETURN TYPE FLAG AND GETTING A NEW "DBR",
   ; THE SS REGISTER OR STACK SEGMENT REGISTER OF THE
   ; PROCESS SELECTED TO RUN NEXT

   INTJOIN: MOV SS:WORD PTR 0,SP      ; SAVE "CURRENT" SP
      MOV SS:WORD PTR 2,BP            ; SAVE 'CURRENT' BP
      MOV SS:WORD PTR 4,CX            ; SAVE IRET_IND FLAG
      CALL GETWORK
      MOV SS,AX                       ; SS:= 'SELECTED' SS_REG

      ;  SWAP VIRTUAL PROCESSOR CONTEXT COMPLETE AT THIS POINT
      ;  NOW OPERATING IN NEWLY SELECTED PROCESS STACK
```

```
        MOV SP,SS:WORD PTR 0      ; SP:= "SELECTED" SP
        MOV BP,SS:WORD PTR 2      ; BP:= "SELECTED" BP
        MOV CX,SS:WORD PTR 4

        ;  CHECK FOR RETURN TYPE, NORMAL OR INTERRUPT

        CMP CX,77H
        JZ  INTRET

    NORM_RET: POP DS
        ; UNLOCK GLOBAL$LOCK
        MOV  AX,SEG GLOBALLOCK
        MOV  ES, AX
        MOV  ES:GLOBALLOCK,0

        STI
        RET

VPSCHEDULER ENDP

;*****************************************************************



;*****************************************************************
;*  INTERRUPT HANDLER                                            *
;*                                                               *

INTERRUPT_HANDLER  PROC NEAR

    ASSUME CS:SCHEDULER
    ASSUME DS:NOTHING
    ASSUME SS:NOTHING
    ASSUME ES:NOTHING

INTVEC: CLI
    PUSH ES     ; SAVE NEEDED REGs TO TEST INTERRUPT FLAG
    PUSH BX
    PUSH AX
    PUSH CX
    CALL HARDWARE_INT_FLAG
    MOV  AL,0
    XCHG  AL,ES:HDWINTFLAG[BX]
    CMP  AL,77H             ; IS INT FLAG ON ?
    JZ   PUSH_REST_REGS     ; IF 'YES' SAVE REST REGs
```

```
    POP  CX                    ; IF 'NOT' RESUME PREVIOUS
    POP  AX                    ; EXECUTION POINT
    POP  BX
    POP  ES
    STI
    IRET

PUSH_REST_REGS: PUSH DX        ; FLAG WAS ON SO NEED
    PUSH DS                    ; RE-SECHEDULE
    PUSH SI
    PUSH DI
    MOV AX,SEG GLOBALLOCK
    MOV  ES, AX
CK: MOV AL,119                  ; LOCK GLOBAL LOCK
    LOCK XCHG ES:GLOBALLOCK,AL
    TEST AL,AL
    JNZ CK

    CALL RDYTHISVP

    MOV CX,77H                ; JUMP TO SCHEDULER
    JMP  INTJOIN

INTRET: POP DI
    POP SI                      ; RETURN FOR
    POP DS                      ; PROCESS WHICH
    POP DX                      ; HAD PREVIOUSLY
    POP CX                      ; BEEN INTERRUPTED
        ; UNLOCK GLOBAL$LOCK
    MOV  AX,SEG GLOBALLOCK
    MOV  ES, AX
    MOV  ES:GLOBALLOCK,0

    POP AX
    POP BX
    POP ES
    STI
    IRET

INTERRUPT_HANDLER ENDP

;*****************************************************************
```

```
;*****************************************************************
;*     HARDWARE INTERRUPT FLAG                                   *
;*                                                               *

HARDWARE_INT_FLAG  PROC  NEAR

   ASSUME CS:SCHEDULER
   ASSUME DS:NOTHING
   ASSUME SS:NOTHING
   ASSUME ES:NOTHING

HDW_FLAG: MOV  AX,SEG PRDS
   MOV  ES, AX
   MOV  BX,0H
   MOV  CL,ES:PRDS[BX]     ;GET CPU #
   MOV  CH,0               ; RETURN IN BX
   MOV  BX,CX
   MOV  AX,SEG HDWINTFLAG  ;SET UP HDW$INT$FLAG
   MOV  ES, AX             ;       SEGMENT
   RET                     ; RETURN IN ES REG

   HARDWARE_INT_FLAG  ENDP

   SCHEDULER ENDS
END
```

APPENDIX H

GLOBAL DATA BASE AND INITIAL PROCESS CODE

Two files are presented here: GLOBAL.SRC and INITK.SRC. They are both separately compiled with the LARGE attribute. They are linked into the file: KORE.LNK. They are represented in the memory map for KORE located at the end of Appendix E. INITK reserves only the minimum amount of space required by operating system for the initial process. The user's initial process may be larger. The user may have to explicitly reserve more space in the LOC86 command. See the file: LOCP.CSD for an example. If you look at the KORE memory map at the end of Appendix F, you will see that the INITMOD_CODE segment is only 25H bytes long.

```
/*****************************************************************/
/*****************************************************************/
/*0009***********************************************************/
/*  FILE:           GLOBAL.SRC
    VERSION:        KLINEF 5-25-82
    PROCEDURES
        DEFINED:    NONE

    REMARKS:THIS MODULE CONTAINS DECLARATIONS FOR ALL THE
            GLOBAL DATA THAT RESIDES IN SHARED COMMON
            MEMORY.  IT'S LOCATED THERE BY THE LOCATE COM-
            MAND AND BY SPECIFYING THAT THE
            GLOBAL$MODULE DATA SEGMENT BE LOCATED AT SOME
            ABSOLUTE ADDRESS.
                                                            */
/*****************************************************************/

GLOBAL$MODULE:  DO;

/*****************************************************************/
/*****************************************************************/
/*  THE FOLLOWING THREE LITERAL DECLARATIONS ARE ALSO          */
/*  GIVEN IN THE LEVEL1 & LEVEL2 MODULES OF THE OPERATING      */
/*  SYSTEM.  A CHANGE HERE WOULD HAVE TO BE REFLECTED IN       */
/*  THOSE MODULES ALSO.                                        */

DECLARE
   MAX$CPU                  LITERALLY  '10',
   MAX$VPS$CPU              LITERALLY  '10',
   MAX$CPU$$$$MAX$VPS$CPU LITERALLY '100';

DECLARE
   GLOBAL$LOCK BYTE PUBLIC INITIAL(0);

/*  THIS SHOULD REFLECT THE MAX$CPU ABOVE */
DECLARE
       NR$RPS           BYTE PUBLIC INITIAL(0),
       NR$VPS(MAX$CPU)  BYTE PUBLIC
                        INITIAL(0,0,0,0,0,0,0,0,0,0);

DECLARE HDW$INT$FLAG(MAX$CPU) BYTE PUBLIC;


DECLARE EVENTS  BYTE  PUBLIC INITIAL(1);

DECLARE EVC$TBL(100) STRUCTURE
        (EVC$NAME     BYTE,
         VALUE        WORD,
         THREAD       BYTE)  PUBLIC
                             INITIAL(0FEH,0,255);
   /* EVC 'FE' IS RESERVED FOR THE OP SYS  */
```

```
DECLARE CPU$INIT BYTE PUBLIC INITIAL(0);

DECLARE SEQUENCERS      BYTE  PUBLIC INITIAL(0);

DECLARE SEQ$TABLE(100) STRUCTURE
        (SEQ$NAME       BYTE,
         SEQ$VALUE      WORD) PUBLIC;

DECLARE VPM( MAX$CPU$$$$MAX$VPS$CPU ) STRUCTURE
        (VP$ID                  BYTE,
         VP$STATE               BYTE,
         VP$PRIORITY            BYTE,
         EVC$THREAD             BYTE,
         EVC$AW$VALUE           WORD,
         SS$REG                 WORD)  PUBLIC;


 END; /* MODULE */

/*****************************************************************/
```

```
/*************************************************************/
/*   INITK      MODULE                          KLINEF 5-27-82*/
/*-----------------------------------------------------------*/
/* THE CODE SEGMENT OF THIS MODULE IS WHAT RESERVES SPACE */
/* BY THE OS FOR THE USER INITIAL PROCESS.  THIS IS       */
/* EXECUTABLE IN IT'S OWN RIGHT.  THUS IF THE USER DOES   */
/* NOT PROVIDE AN INITIAL PROCESS THIS ONE WILL EXECUTE,  */
/* BLOCK ITSELF, AND IDLE THE CPU.  THE ADDRESS OF THE    */
/* INITIAL CODE SEGMENT IS PROVIDED TO LEVEL1 AND IT IS   */
/* REFLECTED IN THE PLM LOCATE COMMAND.  THE ADDRESSES    */
/* PROVIDED MUST AGREE.  THIS PROCESS HAS THE HIGHEST     */
/* PRIORITY AND WILL ALWAYS BE SCHEDULED FIRST BY THE     */
/* SCHEDULER.                                             */
/*-----------------------------------------------------------*/
/* CALLS MADE TO:   AWAIT                                 */
/*************************************************************/

INIT$MOD: DO;

   DECLARE
      MSG13(*) BYTE INITIAL(10,'ENTERING INITIAL PROCESS ',
                                 13,10,'%');

   OUT$LINE: PROCEDURE( PTR ) EXTERNAL;
      DECLARE PTR POINTER;
   END;

   AWAIT:  PROCEDURE( NAME, VALUE ) EXTERNAL;
      DECLARE NAME BYTE, VALUE WORD;
   END;

   INITIAL$PROC: PROCEDURE PUBLIC;

      DECLARE I BYTE;

      /* AFTER INITIALIZATION THIS PROCESS BLOCKS      */
      /* ITSELF TO ALLOW THE NEWLY CREATED PROCESSES   */
      /* TO BE SCHEDULED.                              */
      /* THIS AREA SHOULD BE WRITTEN OVER BY USER INIT */
      /* PROCEDURE MODULE.                             */

      CALL OUT$LINE(@MSG13);

      CALL AWAIT( 0FEH, 1);

   END;   /* INITIAL$PROC */
END;  /* INIT$MOD */
```

APPENDIX I

USER GATE MODULE SOURCE CODE

This source code is contained in file: GATE.SRC. It is compiled with the attribute LARGE. The object code file: GATE.OBJ is given to the user to link with his initial process and user processes.

```
/*0007*************************************************************/
/*  GATE MODULE   FILE 'GATE.SRC'        KLINNEF 5-26-82  */
/*---------------------------------------------------------------*/
/*  THIS MODULE IS GIVEN TO THE USER IN OBJ FORM TO LINK  */
/*  WITH HIS INITIAL AND PROCESS MODULES.  ANY CHANGES TO */
/*  USER SERVICES AVAILABLE FROM THE OS HAVE TO BE        */
/*  REFLECTED HERE.  IN THIS WAY THE USER DOES NOT HAVE TO*/
/*  BE CONCERNED WITH ACTUAL GATEKEEPER SERVICES CODES.   */
/*---------------------------------------------------------------*/
/* ALL CALLS ARE MADE TO THE GATEKEEPER IN LEVEL2 OF THE  */
/* OS.  THE ADDRESS OF THE GATEKEEPER MUST BE GIVEN BELOW.*/
/*0018*************************************************************/

GATE$MOD:  DO;

/* REFLECT GATEKEEPER ADDRESS HERE. GK1=OFFSET, GK2=BASE  */
   DECLARE (GK1,GK2) WORD DATA(0060H,0182H),
                  GATE$KEEPER POINTER AT(@GK1);

   DECLARE RETS WORD,
          RETS$PTR POINTER DATA(@RETS);

/*0029*************************************************************/

   AWAIT:  PROCEDURE( NAME, COUNT ) PUBLIC;
      DECLARE   NAME BYTE,    COUNT WORD;
      CALL  GATE$KEEPER( 0, NAME, COUNT, 0, 0);
      RETURN;
   END;     /* AWAIT */

/*0037*************************************************************/

   ADVANCE: PROCEDURE( NAME ) PUBLIC;
      DECLARE NAME BYTE;
      CALL  GATE$KEEPER( 1, NAME, 0, 0, 0 );
      RETURN;
   END;      /* ADVANCE */

/*0045*************************************************************/

   CREATE$EVC:  PROCEDURE( NAME ) PUBLIC;
      DECLARE NAME BYTE;
      CALL  GATE$KEEPER( 2, NAME, 0, 0, 0 );
      RETURN;
   END;     /* CREATE$EVC */
```

```
/*0075*************************************************************/

   CREATE$SEQ:  PROCEDURE( NAME ) PUBLIC;
      DECLARE NAME BYTE;
      CALL GATE$KEEPER( 3, NAME, 0, 0, 0 );
      RETURN;
   END;     /*  CREATE$SEQ  */

/*0081*************************************************************/

   TICKET:   PROCEDURE( NAME ) WORD PUBLIC;
      DECLARE NAME BYTE;
      CALL GATE$KEEPER( 4, NAME, 0, RETS$PTR );
      RETURN RETS;
   END;     /*   TICKET  */

/*0089*************************************************************/

   READ:     PROCEDURE( NAME ) WORD PUBLIC;
      DECLARE NAME BYTE;
      CALL GATE$KEEPER( 5, NAME, 0, RETS$PTR );
      RETURN RETS;
   END;      /*   READ   */

/*0097*************************************************************/

CREATE$PROC:  PROCEDURE( PROC$ID, PROC$PRI, PROC$STACK$LOC,
                         PROC$IP, PROC$CS ) PUBLIC;
   DECLARE ( PROC$ID, PROC$PRI ) BYTE,
     (PROC$STACK$LOC, PROC$IP, PROC$CS ) WORD;
   DECLARE PROC$TABLE STRUCTURE
     (PROC$ID             BYTE,
      PROC$PRI            BYTE,
      PROC$STACK$SEG      WORD,
      PROC$IP             WORD,
      PROC$CS             WORD);
   DECLARE PROC$PTR POINTER DATA(@PROC$TABLE);

   PROC$TABLE.PROC$ID = PROC$ID;
   PROC$TABLE.PROC$PRI = PROC$PRI;
   PROC$TABLE.PROC$STACK$SEG = PROC$STACK$LOC / 10H;
   PROC$TABLE.PROC$IP = PROC$IP;
   PROC$TABLE.PROC$CS = PROC$CS;
   CALL GATE$KEEPER( 6, 0, 0, PROC$PTR );
   RETURN;
END;   /*  CREATE$PROC  */
```

```
/*0139*************************************************************/

OUT$CHAR: PROCEDURE( CHAR ) PUBLIC;
   DECLARE CHAR BYTE;
   CALL GATE$KEEPER( 8, CHAR, 0, 0, 0);
   RETURN;
END;

/*0147*************************************************************/

OUT$LINE: PROCEDURE( LINE$PTR ) PUBLIC;
   DECLARE LINE$PTR POINTER;
   /* LINE MUST END WITH '%'       */
   /* AND BE LESS THAN 80 CHAR     */
   CALL GATE$KEEPER( 9, 0, 0, LINE$PTR);
   RETURN;
END;

/*0157*************************************************************/

OUT$NUM: PROCEDURE( NUM ) PUBLIC;    /** NUM IS BYTE **/
   DECLARE NUM BYTE;
   CALL GATE$KEEPER( 10, NUM, 0, 0, 0 );
   RETURN;
END;

/*0165*************************************************************/

OUT$DNUM: PROCEDURE( DNUM ) PUBLIC;  /** DNUM IS WORD **/
   DECLARE DNUM WORD;              /** DOUBLE BYTE **/
   CALL GATE$KEEPER( 11, 0, DNUM, 0, 0 );
   RETURN;
END;

/*0173*************************************************************/

PREEMPT: PROCEDURE( PROC$ID ) PUBLIC;
   DECLARE PROC$ID BYTE;
   CALL GATE$KEEPER( 7, PROC$ID, 0, 0, 0 );
   RETURN;
END;  /* PREEMPT */

/*0181*************************************************************/

IN$CHAR:  PROCEDURE (RET$PTR) PUBLIC;
   DECLARE RET$PTR POINTER;
   CALL GATEKEEPER(12, 0, 0, RET$PTR);
END;
```

```
/*0205*************************************************************/

IN$NUM:  PROCEDURE (RET$PTR) PUBLIC;
   DECLARE RET$PTR POINTER;
   CALL GATEKEEPER(13, 0, 0, RET$PTR);
END;

/*0212*************************************************************/

IN$DNUM:  PROCEDURE (RET$PTR) PUBLIC;
   DECLARE RET$PTR POINTER;
   CALL GATEKEEPER(14, 0, 0, RET$PTR);
END;

/*************************************************************/

END;   /* GATE$MOD */

/*************************************************************/
/*************************************************************/
/*************************************************************/
```

APPENDIX J

USER PROCESSES I

This appendix has three parts. It contains the source code for one of the user initial processes, INIT1.SRC. The source code in file, PROC1.SRC, is provided. They are separately compiled with the LARGE attribute. They are linked into P01.LNK and the memory map from file, P01.MP2, is provided. The executable code file P01 is loaded onto one of the SBC's.

```
/*************************************************************/
/*  INITIAL PROCESSS        P-1           KLINEF 5-27-82  */
/*-----------------------------------------------------------*/
/*  THE CODE SEGMENT OF THIS MODULE WILL BE OVERLAYED ON  */
/*  A SEGMENT RESERVED BY THE OS.  THIS INITIAL PROCESS   */
/*  WILL CREATE ALL THE PROCESS THAT WILL RESIDE ON A REAL*/
/*  PROCESSOR.   ALL CALL TO CREATE$PROCESS MUST BE MADE  */
/*  FOR EACH PROCESS.                                     */
/*-----------------------------------------------------------*/
/*  CALLS MADE TO:  CREATE$PROC & AWAIT(TO BLOCK INIT)    */
/*************************************************************/

INIT$MODULE:  DO;

CREATE$PROC: PROCEDURE( PROC$ID, PROC$PRI,
                PROC$STACK$LOC, PROC$IP, PROC$CS ) EXTERNAL;

   DECLARE ( PROC$ID, PROC$PRI ) BYTE,
     (PROC$STACK$LOC, PROC$IP, PROC$CS) WORD;
         END;

   AWAIT: PROCEDURE( EVC$ID, VALUE ) EXTERNAL;
      DECLARE EVC$ID BYTE,
              VALUE  WORD;
   END;

INIT: PROCEDURE PUBLIC;

/*************************************************************/
/*  *   *   * USER AREA OF RESPONSIBILITY  *  *  *  *  * */
/*  MUST MAKE CALL FOR EACH PROCESS TO BE CREATED.  USER  */
/*  PROVIDES ALL PARAMETERS.                              */
/*************************************************************/


CALL CREATE$PROC( 1, 253, 7000H, 06H, 720H );



/*************************************************************/
/*        * * *  END OF USER RESPONSIBLITY  * * *          */
/*************************************************************/

/* THIS STATEMENT BLOCKS THIS PROCESS AND ALLOWS */
/* THE NEWLY CREATED PROCESSES TO BE SCHEDULED.  */

CALL AWAIT( 0FEH, 1 );

END;  /* PROCEDURE */

END;  /*  MODULE  */
```

```
/*****************************************************************/
/*****************************************************************/
/*  PARAMETERS:

PROC$ID:    (BYTE) DESIRED ID FOR PROCESS. 'FE' IS RESERVED.
PROC$PRI:   (BYTE) DESIRED PRIORITY.  HIGH IS 0.
                                      LOW  IS 255.
PROC$STACK$LOC: (WORD) ABSOLUTE ADDRESS OF STACK.  MUST HAVE
            ACCESS TO 120H BYTES OF FREE SPACE.
            USER PROCESS MUST BE CODED IN PROCEDURE BLOCK.
            THE OS PROVIDES STACK OF 110H BYTES.
PROC$IP:    (WORD) USER PROCESS STARTING ADDRESS OFFSET.
PROC$CS:    (WORD) USER PROCESS STARTING ADDRESS BASE.    */
/*****************************************************************/
```

```
/*************************************************************/
/*  PROCESS 1 MODULE                          KLINEF 6-1-82  */
/*-----------------------------------------------------------*/
/*  THIS IS BASED ON THE ORIGINAL DEMONSTRATION THAT COX     */
/*  RAN.                                                     */
/*                                                           */
/* IMPORTANT!  AFTER LINKING AND LOCATING THIS MODULE,       */
/* THE ABSOLUTE ADDRESS OF THE PROCEDURE BLOCK LABEL,        */
/* 'P1' MUST BE REFLECTED IN THE INITIAL PROCESS IN FILE:    */
/* INIT1.SRC                                                 */
/*************************************************************/

CSUPP$MODULE: DO;

   DECLARE     I   WORD;
   DECLARE     CR  LITERALLY '0DH',
               LF  LITERALLY '0AH';

   DECLARE K WORD;

   DECLARE CSUPP BYTE DATA(33);
   DECLARE FLDES BYTE DATA(44);
   DECLARE NEW BYTE DATA(99H);

/*************************************************************/

   DECLARE
      MSG1(*) BYTE INITIAL('PROC #1.  INITIAL ENTRY INTO',
         'CLUTTER SUPPRESSION',10,13,'%'),
      MSG2(*) BYTE INITIAL('PROC #1.  WAIT FOR DATA READY',
         10,13,'%'),
       MSG3(*) BYTE INITIAL('PROC #1.  PERFORMING CLUTTER',
          ' SUPPRESSION:  FRAME # %'),
       MSG4(*) BYTE INITIAL('PROC #1.  ADVANCE FILTER ',
          'DESIGN EVENT COUNT',10,13,'%'),
       MSG5(*) BYTE INITIAL('PROC #1.  CALLED READ TWICE.',
          '  RETURNED: %'),
       MSG6(*) BYTE INITIAL('PROC #1.  CALLED TICKET',
          ' TWICE.  SECOND VALUE RETURNED:  %'),
       MSG7(*) BYTE INITIAL(10,13,'%');
```

```
/*************************************************************/

AWAIT: PROCEDURE(EVC$ID,AWAITED$VALUE) EXTERNAL;
   DECLARE EVC$ID  BYTE,
       AWAITED$VALUE WORD;
END;

ADVANCE: PROCEDURE(EVC$ID) EXTERNAL;
   DECLARE EVC$ID BYTE;
END;

CREATE$EVC: PROCEDURE( NAME ) EXTERNAL;
   DECLARE NAME BYTE;
END;

CREATE$SEQ: PROCEDURE( NAME ) EXTERNAL;
   DECLARE NAME BYTE;
END;

TICKET:  PROCEDURE( NAME ) WORD EXTERNAL;
   DECLARE NAME BYTE;
END;

READ:  PROCEDURE( NAME ) WORD EXTERNAL;
   DECLARE NAME BYTE;
END;

OUT$CHAR: PROCEDURE(CHAR) EXTERNAL;
   DECLARE CHAR BYTE;
END;

OUT$NUM: PROCEDURE( NUM ) EXTERNAL;
   DECLARE NUM BYTE;
END;

OUT$DNUM: PROCEDURE( DNUM ) EXTERNAL;
   DECLARE DNUM WORD;
END;

OUT$LINE: PROCEDURE( LINE$PTR ) EXTERNAL;
   DECLARE LINE$PTR POINTER;
END;
```

```
/*************************************************************/

P1: PROCEDURE PUBLIC;

   CALL OUT$LINE(@MSG1);

   /*  CREATE SYNCHRONIZATION PRIMITIVES  */
   CALL CREATE$EVC(CSUPP);
   CALL CREATE$EVC(FLDES);

   /*  CALL READ AS A TEST  */
   K = READ( CSUPP);
   CALL OUT$LINE( @MSG5 );
   CALL OUT$DNUM( K );
   CALL OUT$LINE(@MSG7);

   /*  CREATE SEQUENCER PRIMITIVES  */
   CALL CREATE$SEQ(NEW);
   /*  CALL TICKET TWICE AS A TEST  */
   K = TICKET(NEW);
   K = TICKET(NEW);

   CALL OUT$LINE(@MSG6);
   CALL OUT$DNUM( K );
   CALL OUT$LINE(@MSG7);

   /*************************************************************/
   /* PROCESS 1 LOOP BEGINS HERE                                */

   I = 1;
   DO WHILE (I <= 0FFFFH);
      CALL OUT$LINE(@MSG2);

      CALL AWAIT(CSUPP,I);
      I = I + 1;     /*  <========  */

      CALL OUT$LINE(@MSG3);
      CALL OUT$DNUM(I);
      CALL OUT$LINE(@MSG7);

      DO K = 0 TO 1000;
         CALL TIME(250);
      END;

      CALL OUT$LINE(@MSG4);

      CALL ADVANCE(FLDES);

   END; /* WHILE */
END;  /* P1 */
END; /* MODULE */
```

```
ISIS-II MCS-86 LOCATER, V1.1 INVOKED BY:
LOC86 P01.LNK ADDRESSES(SEGMENTS(&
INITMODULE_CODE(02800H),&
CSUPPMODULE_CODE(07200H),&
CSUPPMODULE_DATA(07500H),&
GATEMOD_CODE(07700H),&
GATEMOD_DATA(079E0H),&
STACK(07000H)))&
SEGSIZE(STACK(100H))&
RS(0 TO 27FFH)

SYMBOL TABLE OF MODULE INITMODULE
READ FROM FILE P01.LNK
WRITTEN TO FILE :F0:P01

BASE   OFFSET TYPE SYMBOL        BASE   OFFSET TYPE SYMBOL

0280H  0002H  PUB  INIT          0720H  0006H  PUB  P1
0770H  022DH  PUB  INDNUM        0770H  020CH  PUB  INNUM
0770H  01EBH  PUB  INCHAR        0770H  01C8H  PUB  PREEMPT
0770H  01A8H  PUB  OUTDNUM       0770H  0185H  PUB  OUTNUM
0770H  0164H  PUB  OUTLINE       0770H  0141H  PUB  OUTCHAR
0770H  00F4H  PUB  CREATEPROC    0770H  00C8H  PUB  READ
0770H  009CH  PUB  TICKET        0770H  0079H  PUB  CREATESEQ
0770H  0056H  PUB  CREATEEVC     0770H  0033H  PUB  ADVANCE
0770H  000EH  PUB  AWAIT

MEMORY MAP OF MODULE INITMODULE
READ FROM FILE P01.LNK
WRITTEN TO FILE :F0:P01

SEGMENT MAP

START     STOP      LENGTH ALIGN NAME              CLASS

02800H    02830H     0031H   W   INITMODULE_COD   CODE
                                 -E
02832H    02832H     0000H   W   INITMODULE_DAT   DATA
                                 -A
07000H    070FFH     0100H   W   STACK            STACK
07200H    0732DH     012EH   W   CSUPPMODULE_CO   CODE
                                 -DE
07500H    0761CH     011DH   W   CSUPPMODULE_DA   DATA
                                 -TA
07700H    0794DH     024EH   W   GATEMOD_CODE     CODE
079E0H    079E9H     000AH   W   GATEMOD_DATA     DATA
079EAH    079EAH     0000H   W   MEMORY           MEMORY
```

APPENDIX K

USER PROCESSES II

This appendix has three parts. It contains the source code for one of the user initial processes, INIT2.SRC. The source code for user processes in files, PROC2.SRC, PROC3.SRC, and PROC5.SRC is provided. They are all linked into file, P02.LNK. The executable code file, P02 is loaded onto one of the SBC's. The memory map in file, P02.MP2, is also provided.

```
/*************************************************************/
/* INITIAL PROCESS    P-2,3,5              KLINEF 6-1,82     */
/*-----------------------------------------------------------*/
/*  THE CODE SEGMENT OF THIS MODULE WILL BE OVERLAYED ON     */
/*  A SEGMENT RESERVED BY THE OS.  THIS INITIAL PROCESS      */
/*  WILL CREATE ALL THE PROCESSES THAT WILL RESIDE ON A      */
/*  REAL PROCESSOR.   ALL CALLS TO CREATE$PROCESS MUST BE    */
/*  MADE FOR EACH PROCESS.                                   */
/*-----------------------------------------------------------*/
/*  CALLS MADE TO:  CREATE$PROC & AWAIT(TO BLOCK INIT)       */
/*************************************************************/

INIT$MODULE:  DO;

CREATE$PROC: PROCEDURE( PROC$ID, PROC$PRI,
                PROC$STACK$LOC, PROC$IP, PROC$CS ) EXTERNAL;

   DECLARE ( PROC$ID, PROC$PRI ) BYTE,
     (PROC$STACK$LOC, PROC$IP, PROC$CS) WORD;
         END;

   AWAIT: PROCEDURE( EVC$ID, VALUE ) EXTERNAL;
      DECLARE EVC$ID BYTE,
              VALUE  WORD;
   END;

INIT: PROCEDURE PUBLIC;

/*************************************************************/
/*  *   *   *  USER AREA OF RESPONSIBILITY  *  *  *  *  *  */
/*  MUST MAKE CALL FOR EACH PROCESS TO BE CREATED.  USER     */
/*  PROVIDES ALL PARAMETERS.                                 */
/*************************************************************/


CALL CREATE$PROC( 2, 245, 7000H, 04H, 720H );
CALL CREATE$PROC( 3, 253, 9000H, 04H, 920H );
CALL CREATE$PROC( 5, 254, 8000H, 06H, 820H );

/*************************************************************/
/*         * * *  END OF USER RESPONSIBLITY  * * *          */
/*************************************************************/

/* THIS STATEMENT BLOCKS THIS PROCESS AND ALLOWS */
/* THE NEWLY CREATED PROCESSES TO BE SCHEDULED.  */

CALL AWAIT( 0FEH, 1 );

END;  /* PROCEDURE */

END;  /*  MODULE  */
```

```
/*****************************************************************/
/*****************************************************************/
/*  PARAMETERS:

PROC$ID:     (BYTE) DESIRED ID FOR PROCESS. 'FE' IS RESERVED.
PROC$PRI:    (BYTE) DESIRED PRICRITY.  HIGH IS 0.
                                       LOW  IS 255.
PROC$STACK$LOC: (WORD) ABSOLUTE ADDRESS OF STACK.  MUST HAVE
             ACCESS TO 120H BYTES OF FREE SPACE.
             USER PROCESS MUST BE CODED IN PROCEDURE BLOCK.
             THE OS PROVIDES STACK OF 110H BYTES.
PROC$IP:     (WORD) USER PROCESS STARTING ADDRESS OFFSET.
PROC$CS:     (WORD) USER PROCESS STARTING ADDPESS BASE.    */
/*****************************************************************/
```

```
/*************************************************************/
/* PROCESS 2 MODULE                              KLINEF 6-2-82  */
/*-----------------------------------------------------------*/
/* THIS  IS BASED ON THE ORIGINAL DEMONSTRATION THAT COX     */
/* RAN.                                                      */
/*                                                           */
/* IMPORTANT!  ONCE THIS MODULE IS LINKED AND LOCATED,       */
/* THE ABSOLUTE ADDRESS OF THE PROCEDURE BLOCK LABEL,        */
/* 'P02' MUST BE REFLECTED IN THE INITIAL PROCESS IN FILE:   */
/* INIT2.SRC                                                 */
/*************************************************************/

FLDES$MODULE: DO;

   DECLARE I      WORD;
   DECLARE Z      BYTE;
   DECLARE CSUPP BYTE DATA(33);
   DECLARE FLDES BYTE DATA(44);

/*************************************************************/

   DECLARE
      MSG1(*) BYTE INITIAL('PROC #2.  INITIAL ENTRY INTO',
         ' FILTER DESIGN',13,10,'$'),
      MSG2(*) BYTE INITIAL('PROC #2.  WAIT FOR DATA READY',
         13,10,'$'),
      MSG3(*) BYTE INITIAL('PROC #2.  PERFORMING FILTER ',
         'DESIGN ON FRAME # $'),
      MSG4(*) BYTE INITIAL('PROC #2.  ADVANCE CLUTTER ',
         'SUPPRESSION EVENT COUNT',10,13,'$'),
      MSG5(*) BYTE INITIAL(10,13,'$');

/*************************************************************/

AWAIT: PROCEDURE(EVC$ID,AWAITED$VALUE) EXTERNAL;
   DECLARE EVC$ID BYTE,
      AWAITED$VALUE WORD;
END;

ADVANCE: PROCEDURE(EVC$ID) EXTERNAL;
   DECLARE EVC$ID BYTE;
END;

CREATE$EVC: PROCEDURE(NAME) EXTERNAL;
   DECLARE NAME BYTE;
END;

OUT$LINE: PROCEDURE(PTR) EXTERNAL;
   DECLARE PTR POINTER;
END;
```

```
OUT$DNUM: PROCEDURE(DNUM) EXTERNAL;
   DECLARE DNUM WORD;
END;

/*************************************************************/

P2: PROCEDURE PUBLIC;

   CALL OUT$LINE(@MSG1);

    /*  CREATE THE SYNCHRONIZATION PRIMITIVES  */
   CALL CREATE$EVC(CSUPP);
   CALL CREATE$EVC(FLDES);

   /*************************************************************/
   /* BEGIN PROCESS 2 LOOP  HERE                                */
   I = 0;
   DO WHILE (I <= 0FFFFH);

      CALL OUT$LINE(@MSG2);

      CALL AWAIT(FLDES,I);
      I = I + 1;

      CALL OUT$LINE(@MSG3);
      CALL OUT$DNUM(I);
      CALL OUT$LINE(@MSG5);

      DO Z = 0 TO 100;
         CALL TIME(250);
      END;

      CALL OUT$LINE(@MSG4);

      CALL ADVANCE(CSUPP);

   END; /* WHILE */
END;    /* P2 */
END; /*MODULE */
```

```
/*****************************************************************/
/* PROCESS 3  MODULE                              KLINEF 6-2-82  */
/*---------------------------------------------------------------*/
/* IMPORTANT!  ONCE THIS MODULE IS LINKED AND LOCATED,           */
/* THE ABSOLUTE ADDRESS OF THE PROCEDURE BLOCK LABEL,            */
/* 'P03' MUST BE REFLECTED IN THE INITIAL PROCESS IN FILE:       */
/* INIT2.SRC.                                                    */
/*****************************************************************/

FLDES3$MODULE: DO;

   DECLARE     I   WORD;
   DECLARE     CR  LITERALLY '0DH',
               LF  LITERALLY '0AH';
   DECLARE     Z BYTE;
   DECLARE     FLDES BYTE DATA(44);

/*****************************************************************/

   DECLARE
      MSG1(*) BYTE INITIAL('PROC #3.  INITIAL ENTRY',
          ' INTO FILTER DESIGN PART 2',10,13,'%'),
      MSG2(*) BYTE INITIAL('PROC #3.  WAIT FOR DATA',
          ' READY',10,13,'%'),
      MSG3(*) BYTE INITIAL('PROC #3.  PERFORMING PART 2',
          ' FILTER DESIGN ON FRAME # %'),
      MSG4(*) BYTE INITIAL(10,13,'%');

/*****************************************************************/

AWAIT: PROCEDURE(EVC$ID,AWAITED$VALUE) EXTERNAL;
   DECLARE EVC$ID BYTE,
      AWAITED$VALUE WORD;
END;

ADVANCE: PROCEDURE(EVC$ID) EXTERNAL;
   DECLARE EVC$ID BYTE;
END;

CREATE$EVC: PROCEDURE(NAME) EXTERNAL;
   DECLARE NAME BYTE;
END;

OUT$LINE: PROCEDURE(PTR) EXTERNAL;
   DECLARE PTR POINTER;
END;

OUT$DNUM: PROCEDURE(DNUM) EXTERNAL;
   DECLARE DNUM WORD;
END;
```

```
/*************************************************************/

P3: PROCEDURE PUBLIC;

   CALL OUT$LINE(@MSG1);

   /*  CREATE THE SYNCHRONIZATION PRIMITIVES  */
   CALL CREATE$EVC(FLDES);

   /*************************************************************/
   /* PROCESS 3 LOOP BEGINS HERE                                */

   I = 1;
   DO WHILE (I <= 0FFFFH);

      CALL OUT$LINE(@MSG2);

      CALL AWAIT(FLDES,I);
      I = I + 1;

      CALL OUT$LINE(@MSG3);
      CALL OUT$DNUM(I);
      CALL OUT$LINE(@MSG4);

      DO Z = 0 TO 50;
         CALL TIME(250);
      END;

   END; /* WHILE */
END;    /* P3 */
END; /*MODULE */
```

```
/*************************************************************/
/*   PROCESS 5  MODULE                          KLINEF 6-2-82 */
/*-----------  ------------------------------------------------*/
/* THIS PROCESS ACCOMPLISHES TESTS OF THE OPERATING SYSTEM*/
/* NOT YET DONE.  ALL OF THE NEWLY ADDED INPUT SERVICES   */
/* WILL BE TESTED HERE BEFORE THE PROCESS GOES INTO IT'S  */
/* LOOP.  THIS INCLUDES PREEMT.                           */
/* THIS PROCESS ACTS ON DATA LOCATED IN A BUFFER IN       */
/* SHARED COMMMON MEMORY.  PROCESS 4 , RESIDENT IN SOME   */
/* OTHER CPU, WILL OUTPUT THE DATA AFTER IT HAS BEEN PRO- */
/* CESSED.  THE EFFECT WILL BE EVIDENT ON THE SCREEN IF   */
/* THEY COORDINATE CORRECTLY.                             */
/*                                                        */
/* IMPORTANT!  AFTER LINKING AND LOCATING THIS MODULE,    */
/* MAKE SURE THE ABSOLUTE ADDRESS OF THE PROCEDURE BLOCK  */
/* LABEL, 'P05', HAS BEEN REFLECTED IN THE INITIAL PRO-   */
/* CESS IN FILE INIT2.WRK                                 */
/*************************************************************/

PROC5$MODULE: DO;

DECLARE
  I                   BYTE, K WORD,
  FOREVER             LITERALLY '0FFH',
  MONITOR             LITERALLY '0FEH',
  INCHR               BYTE,
  WORD$VALUE          WORD,
  BYTE$VALUE          BYTE,
  GLOBAL$BUFFER(70)   BYTE AT (0E0700H),
  LOCAL$BUFFER (70)   BYTE,
  PROC4$5$EVC1        BYTE INITIAL (10H),
  PROC4$5$EVC2        BYTE INITIAL (11H);

/*************************************************************/

DECLARE
   MSG1(*) BYTE INITIAL('PROC #5.  INITIAL ENTRY%'),
   MSG2(*) BYTE INITIAL(10,13,'PROC #5.  PROCESSING DATA',
       ' IN SHARED BUFFER%'),
   MSG3(*) BYTE INITIAL(10,13,'PROC #5.  FINISEED PROCESS',
      'ING',13,10,'%'),
   MSG4(*) BYTE INITIAL(10,13,'DO YOU WANT TO PREEMPT THE',
       ' MONITOR(Y OR N)?%'),
   MSG9(*) BYTE INITIAL(10,13,'%');

/*************************************************************/

AWAIT:  PROCEDURE(EVC$ID,AWAITED$VALUE) EXTERNAL;
   DECLARE EVC$ID BYTE, AWAITED$VALUE BYTE;
END;
```

```
ADVANCE:  PROCEDURE(EVC$ID) EXTERNAL;
   DECLARE EVC$ID BYTE;
END;

CREATE$EVC:  PROCEDURE(NAME) EXTERNAL;
   DECLARE NAME BYTE;
END;

CREATE$SEQ:  PROCEDURE(NAME) EXTERNAL;
   DECLARE NAME BYTE;
END;

OUT$CHAR:  PROCEDURE(CHAR) EXTERNAL;
   DECLARE CHAR BYTE;
END;

IN$CHAR:  PROCEDURE(RET$PTR) EXTERNAL;
   DECLARE RET$PTR POINTER;
END;

PREEMPT:  PROCEDURE(VP$ID) EXTERNAL;
   DECLARE VP$ID BYTE;
END;

OUT$LINE:  PROCEDURE(PTR) EXTERNAL;
   DECLARE PTR POINTER;
END;

IN$NUM: PROCEDURE(RET$PTR) EXTERNAL;
   DECLARE RET$PTR POINTER;
END;

IN$DNUM:  PROCEDURE(RET$PTR) EXTERNAL;
   DECLARE RET$PTR POINTER;
END;

OUT$DNUM:PROCEDURE(WORDS) EXTERNAL;
   DECLARE WORDS WORD;
END;

OUT$NUM:   PROCEDURE(BYT)  EXTERNAL;
   DECLARE BYT BYTE;
END;
```

```
/*************************************************************/

P5:  PROCEDURE PUBLIC;

   CALL OUT$LINE(@MSG1);
   CALL CREATE$EVC(PROC4$5$EVC1);
   CALL CREATE$EVC(PROC4$5$EVC2);
   DO I = 0 TO 69;
      GLOBAL$BUFFER(I) = '.';
   END;
   GLOBAL$BUFFER(0) = 'X';
   K = 0;
   /*********************************************************/
   /* PROCESS 5 LOOP BEGINS HERE                            */
   DO WHILE FOREVER;
      CALL AWAIT(PROC4$5$EVC1,K);
      IF K = 35 THEN CALL PREEMPT(MONITOR);
      K = K + 1;
      CALL OUT$LINE(@MSG2);
      DO I = 0 TO 69;
         LOCAL$BUFFER(I) = GLOBAL$BUFFER(I);
      END;
      I = 0;
      DO WHILE LOCAL$BUFFER(I) <> 'X';
         I = I + 1;
      END;
      IF I = 69 THEN LOCAL$BUFFER(0) = 'X';
      ELSE LOCAL$BUFFER(I + 1) = 'X';
      LOCAL$BUFFER(I) = '.';
      DO I = 0 TO 69;
         GLOBAL$BUFFER(I) = LOCAL$BUFFER(I);
      END;
      CALL OUT$LINE(@MSG3);
      CALL ADVANCE(PROC4$5$EVC2);
   END; /* DO FOREVER */
END; /* P5 */
END;  /* PROC5$MODULE */
```

```
ISIS-II MCS-86 LOCATER, V1.1 INVOKED BY:
LOC86 P02.LNK ADDRESSES SEGMENTS(&
INITMODULE_CODE(0280H),&
FLDESMODULE_CODE(07200H),&
FLDESMODULE_DATA(07500H),&
GATEMOD_CODE(07700H),&
GATEMOD_DATA(07900H),&
PROC5MODULE_CODE(09200H),&
PROC5MODULE_DATA(09500H),&
FLDES3MODULE_CODE(09200H),&
FLDES3MODULE_DATA(09500H),&
STACK(07000H)))&
SEGSIZE(STACK(100H))&
RS(0 TO 27FFH)

SYMBOL TABLE OF MODULE INITMODULE
READ FROM FILE P02.LNK
WRITTEN TO FILE :F0:P02

BASE   OFFSET TYPE SYMBOL          BASE   OFFSET TYPE SYMBOL

0280H  0002H  PUB  INIT            0720H  0004H  PUB  P2
0920H  0004H  PUB  P3              0820H  0006H  PUB  P5
0770H  022DH  PUB  INDNUM          0770H  020CH  PUB  INNUM
0770H  01EBH  PUB  INCHAR          0770H  01C8H  PUB  PREEMPT
0770H  01A8H  PUB  OUTDNUM         0770H  0185H  PUB  OUTNUM
0770H  0164H  PUB  OUTLINE         0770H  0141H  PUB  OUTCHAR
0770H  00F4H  PUB  CREATEPROC      0770H  00C8H  PUB  READ
0770H  009CH  PUB  TICKET          0770H  0079H  PUB  CREATESEG
0770H  0056H  PUB  CREATEEVC       0770H  0033H  PUB  ADVANCE
0770H  000EH  PUB  AWAIT

MEMORY MAP OF MODULE INITMODULE
READ FROM FILE P02.LNK
WRITTEN TO FILE :F0:P02
```

SEGMENT MAP

START	STOP	LENGTH	ALIGN	NAME	CLASS
02800H	0285FH	005FH	W	INITMODULE_COD-E	CODE
02860H	02860H	0000H	W	INITMODULE_DAT-A	DATA
07000H	070FFH	0100H	W	STACK	STACK
07200H	072B5H	00B6H	W	FLDESMODULE_CO-DE	CODE
07500H	075B5H	00B6H	W	FLDESMODULE_DA-TA	DATA
07700H	0794DH	024FH	W	GATEMOD_CODE	CODE
079E0H	079E9H	000AH	W	GATEMOD_DATA	DATA
08200H	08341H	0142H	W	PROCSMODULE_CO-DE	CODE
08500H	085E6H	00E7H	W	PROCSMODULE_DA-TA	DATA
09200H	09296H	0097H	W	FLDES3MODULE_C-ODE	CODE
09500H	0958FH	0090H	W	FLDES3MODULE_D-ATA	DATA
E0700H	E0745H	0046H	A	(ABSOLUTE)	
E0746H	E0746H	0000H	W	MEMORY	MEMORY

APPENDIX L

USER PROCESSES III

This appendix has three parts. It contains the source code for one of the user initial processes, INIT3.SRC. The source code for user processes in file, PROC4.SRC, is provided. They are all linked into file, P03.LNK. The executable code file, P03 is loaded onto one of the SBC's. The memory map in file, P03.MP2, is also provided.

```
/*************************************************************/
/*  INITIAL PROCESSS       P-4            KLINEF 5-27-82  */
/*-----------------------------------------------------------*/
/*  THE CODE SEGMENT OF THIS MODULE WILL BE OVERLAYED ON  */
/*  A SEGMENT RESERVED BY THE OS.  THIS INITIAL PROCESS   */
/*  WILL CREATE ALL THE PROCESS THAT WILL RESIDE ON A REAL*/
/*  PROCESSOR.   ALL CALL TO CREATE$PROCESS MUST BE MADE  */
/*  FOR EACH PROCESS.                                     */
/*-----------------------------------------------------------*/
/*  CALLS MADE TO:  CREATE$PROC & AWAIT(TO BLOCK INIT)    */
/*************************************************************/

INIT$MODULE:  DO;

CREATE$PROC: PROCEDURE( PROC$ID, PROC$PRI,
                PROC$STACK$LOC, PROC$IP, PROC$CS ) EXTERNAL;

   DECLARE ( PROC$ID, PROC$PRI ) BYTE,
     (PROC$STACK$LOC, PROC$IP, PROC$CS) WORD;
         END;

   AWAIT: PROCEDURE( EVC$ID, VALUE ) EXTERNAL;
      DECLARE EVC$ID BYTE,
              VALUE  WORD;
   END;

INIT: PROCEDURE PUBLIC;

/*************************************************************/
/*  *   *   * USER AREA OF RESPONSIBILITY  *  *  *  *  * */
/*  MUST MAKE CALL FOR EACH PROCESS TO BE CREATED.  USER  */
/*  PROVIDES ALL PARAMETERS.                              */
/*************************************************************/



   CALL CREATE$PROC(4, 5, 7000H, 06H,  0720H );



/*************************************************************/
/*        * * *  END OF USER RESPONSIBLITY  * * *          */
/*************************************************************/

/* THIS STATEMENT BLOCKS THIS PROCESS AND ALLOWS */
/* THE NEWLY CREATED PROCESSES TO BE SCHEDULED.  */

CALL AWAIT( 0FEH, 1 );

END;  /* PROCEDURE */

END;  /*  MODULE  */
```

```
/*****************************************************************/
/*****************************************************************/
/*  PARAMETERS:

PROC$ID:     (BYTE) DESIRED ID FOR PROCESS. 'FE' IS RESERVED.
PROC$PRI:    (BYTE) DESIRED PRIORITY.  HIGH IS 0.
                                       LOW  IS 255.
PROC$STACK$LOC: (WORD) ABSOLUTE ADDRESS OF STACK.  MUST HAVE
             ACCESS TO 120H BYTES OF FREE SPACE.
             USER PROCESS MUST BE CODED IN PROCEDURE BLOCK.
             THE OS PROVIDES STACK OF 110H BYTES.
PROC$IP:     (WORD) USER PROCESS STARTING ADDRESS OFFSET.
PROC$CS:     (WORD) USER PROCESS STARTING ADDRESS BASE.    */
/*****************************************************************/
```

```
/*****************************************************************/
/*   PROCESS 4  MODULE                              KLINEF 6-2-82 */
/*---------------------------------------------------------------*/
/*  IMPORTANT!  AFTER LINKING AND LOCATING THIS MODULE,          */
/*  MAKE SURE THE ABSOLUTE ADDRESS OF THE PROCEDURE BLOCK        */
/*  LABEL, 'P05', HAS BEEN REFLECTED IN THE INITIAL PRO-         */
/*  CESS IN FILE INIT2.WRK                                       */
/*****************************************************************/

PROC4$MODULE: DO;

DECLARE
 (I,J)                BYTE, K WORD,
  FOREVER             LITERALLY '0FFH',
  GLOBAL$BUFFER(70)   BYTE AT (0E0700H),
  LOCAL$BUFFER (70)   BYTE,
  PROC4$5$EVC1        BYTE INITIAL (10H),
  PROC4$5$EVC2        BYTE INITIAL (11H);

/*****************************************************************/

DECLARE
   MSG9(*) BYTE INITIAL(10,13,'%');

/*****************************************************************/

AWAIT:  PROCEDURE(EVC$ID,AWAITED$VALUE) EXTERNAL;
   DECLARE EVC$ID BYTE, AWAITED$VALUE BYTE;
END;

ADVANCE:  PROCEDURE(EVC$ID) EXTERNAL;
   DECLARE EVC$ID BYTE;
END;

CREATE$EVC:  PROCEDURE(NAME) EXTERNAL;
   DECLARE NAME BYTE;
END;

OUT$CHAR:  PROCEDURE(CHAR) EXTERNAL;
   DECLARE CHAR BYTE;
END;

OUT$LINE:  PROCEDURE(PTR) EXTERNAL;
   DECLARE PTR POINTER;
END;
```

```
/*************************************************************/

P4:  PROCEDURE PUBLIC;

   CALL CREATE$EVC(PROC4$5$EVC1);
   CALL CREATE$EVC(PROC4$5$EVC2);
   /*************************************************************/
   /* PROCESS 5 LOOP BEGINS HERE                                */
   K = 0001H;
   DO WHILE FOREVER;
      CALL AWAIT(PROC4$5$EVC2,K);
      K = K + 1;
      DO I = 0 TO 69;
         LOCAL$BUFFER(I) = GLOBAL$BUFFER(I);
      END;
      DO I = 0 TO 69;
         CALL OUT$CHAR(LOCAL$BUFFER(I));
      END;
      CALL OUT$LINE(@MSG9);
      DO I = 0 TO 255;
         CALL TIME(250);
      END;
      CALL ADVANCE(PROC4$5$EVC1);
   END; /* DO FOREVER */
END; /* P4 */
END;  /* PROC4$MODULE */
```

```
ISIS-II MCS-86 LOCATER, V1.1 INVOKED BY:
LOC86 P03.LNK ADDRESSES(SEGMENTS(&
INITMODULE_CODE(02800H),&
PROC4MODULE_CODE(7200H),&
PROC4MODULE_DATA(7500H),&
STACK(7000H)))&
SEGSIZE(STACK(100H))&
RS(0 TO 71FFH)
WARNING 56:  SEGMENT IN RESERVED SPACE
  SEGMENT:  INITMODULE_CODE
WARNING 56:  SEGMENT IN RESERVED SPACE
  SEGMENT:  STACK

SYMBOL TABLE OF MODULE INITMODULE
READ FROM FILE P03.LNK
WRITTEN TO FILE :F0:P03

BASE   OFFSET TYPE SYMBOL        BASE   OFFSET TYPE SYMBOL

0280H  0002H  PUB  INIT          0720H  0006H  PUB  P4
0755H  022DH  PUB  INDNUM        0755H  020CH  PUB  INNUM
0755H  01EBH  PUB  INCHAR        0755H  01C8H  PUB  PREEMPT
0755H  01A9H  PUB  OUTDNUM       0755H  0185H  PUB  OUTNUM
0755H  0164H  PUB  OUTLINE       0755H  0141H  PUB  OUTCHAR
0755H  00F4H  PUB  CREATEPROC    0755H  00C8H  PUB  READ
0755H  009CH  PUB  TICKET        0755H  0079H  PUB  CREATESEQ
0755H  0056H  PUB  CREATEEVC     0755H  0033H  PUB  ADVANCE
0755H  000EH  PUB  AWAIT

MEMORY MAP OF MODULE INITMODULE
READ FROM FILE P03.LNK
WRITTEN TO FILE :F0:P03

SEGMENT MAP

START     STOP      LENGTH ALIGN NAME              CLASS

02800H    02830H     0031H  W    INITMODULE_COD  CODE
                                 -E
07000H    070FFH     0100H  W    STACK           STACK
07200H    072C2H     00C3H  W    PROC4MODULE_CO  CODE
                                 -DE
072C4H    072C4H     0000H  W    INITMODULE_DAT  DATA
                                 -A
072C4H    072CDH     000AH  W    GATEMOD_DATA    DATA
07500H    0754EH     004FH  W    PROC4MODULE_DA  DATA
                                 -TA
07550H    0779DH     024EH  W    GATEMOD_CODE    CODE
E0700H    E0745H     0046H  A    (ABSOLUTE)
E0746H    E0746H     0000H  W    MEMORY          MEMORY
```

LIST OF REFERENCES

1. INTEL Corporation, The 8086 Family User's Manual, October 1979.

2. INTEL Corporation, iSBC 86/12 Single Board Computer Hardware Reference Manual, 1976.

3. INTEL Corporation, ISIS-II User's Guide, 1978.

4. INTEL Corporation, 8086 Family Utilities User's Guide for 8080/8085 Based Development Systems, 1980.

5. INTEL Corporation, MCS-86 Macro Assembly Language Reference Manual, 1979.

6. INTEL Corporation, MSC-86 Assembler Operating Instructions fo ISIS-II Users, 1978.

7. INTEL Corporation, ISIS-II PL/M-86 Compiler Operator's Manual, 1978.

8. INTEL Corporation, PLM-86 Programming Manual for 8080/8085 Based Development Systems, 1980.

9. Reed, D.P. and Kanodia, R.J., "Synchronization with Eventcounts and Sequencers", Communication of the ACM, v. 22, p. 115-123, February 1979.

10. Wasson, W.J., Detailed Design of the Kernel of a Real-Time Multiprocessor Operating System, M.S. Thesis, Naval Postgraduate School, June 1980.

11. Rapantzikos, D.K., Detailed Design of the Kernel of a Real-Time Multiprocessor Operating System, M.S. Thesis, Naval Postgraduate School, March 1981.

12. Cox, E.R., A Real-Time, Distributed Operating System for a Multiple Computer System, M.S. Thesis, Naval Postgraduate School, December 1981.

13. Saltzer, J.H., Traffic Control in a Multiplexed Computer System, Ph.D Thesis, Massachusetts Institute of Technology, 1966.

14. Guillen, L.A., A Tactical System Emulator for a Distributed Micro-computer Architecture, M.S. Thesis, Naval Postgraduate School, June 1979.

INITIAL DISTRIBUTION LIST

		No. of copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School 93940	1
4.	Dr. M. L. Cotton, Code 62Cc Deparment of Electrical Engineering Naval Postgraduate School Monterey, California 93940	1
5.	Dr. Uno R. Kodres, Code 52Kr Deparment of Computer Science Naval Postgraduate School Monterey, California 93940	3
6.	United States Military Academy AAD - Office of the Dean ATTN: CPT Stephen Klinefelter West Point, New York 10996	2
7.	Daniel Green, Code 20E Naval Surface Weapons Center Dahlgren, Virginia 22449	1
8.	Commander J. Donegan, USN PMS 40DBY Naval Sea Systems Command Washington, D.C. 20362	1